Universidade do Minho
Escola de Engenharia

Carlos Alberto da Cunha Fernandes

Porting e expansão de um µkernel SMP baseado em P-Thread para uma plataforma ARM Quad-core

Porting e expansão de um µkernel SMP baseado em P-Thread para uma plataforma ARM Quad-core
Carlos Alberto da Cunha Fernandes
UMinho | 2015

novembro de 2015

Universidade do Minho
Escola de Engenharia

Carlos Alberto da Cunha Fernandes

Porting e expansão de um μkernel SMP baseado em P-Thread para uma plataforma ARM Quad-core

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Adriano Tavares

novembro de 2015

Declaração

Nome: Carlos Alberto da Cunha Fernandes
Endereço Eletrónico: fernandescacf@gmail.com
Telemóvel:
Bilhete de Identidade/Cartão do Cidadão: 14096493
Título da Dissertação: Porting e expansão de um *µkernel* SMP baseado em P-Thread para uma plataforma ARM Quad-core

Orientador: Professor Doutor Adriano Tavares

Ano de conclusão: 2015
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, _____/_____/_____

Assinatura:_____

Agradecimentos

As minhas primeiras palavras de agradecimento são dirigidas à minha família, por me terem dado a possibilidade de realizar esta dissertação, e por todo o apoio, compreensão, confiança e incentivo que me proporcionaram ao longo do meu percurso académico.

Gostaria também de agradecer ao meu orientador, o Doutor Adriano Tavares, por todo o apoio e orientação que me providenciou durante a realização deste trabalho.

Um especial obrigado ao Mestre Sandro Pinto pela orientação, apoio, motivação, por todos os conhecimentos transmitidos e pela grande disponibilidade que teve no decorrer desta dissertação.

Ao Diogo Lima e ao Raphael Gonçalves, por toda a amizade, cooperação e disponibilidade para ajudar que sempre demonstraram no decurso desta dissertação.

Por fim, gostaria de agradecer a todos os meus amigos e colegas de curso, que me acompanharam ao longo do meu percurso académico.

A todos o meu muito obrigado.

Resumo

Os processadores *multicore* estão em todo o lado e o seu uso nos sistemas embebidos tem vindo a crescer consideravelmente nos últimos anos. Atualmente, os requisitos de um sistema embebido são bastante diferentes do que eram há alguns anos atrás, passando de realizar tarefas bastante simples e específicas (algoritmos de controlo), para realizar tarefas bastante mais complexas e sofisticadas com requisitos de processamento bastante mais elevados (GUIs e Internet). Esta exigência crescente de requisitos a nível aplicacional não foi acompanhada pelos processadores *singlecore*, tornando portanto a migração para processadores *multicore* inevitável.

Contudo, desenvolver aplicações *bare-metal* que tirem proveito das potencialidades destes processadores pode tornar-se uma tarefa complexa e morosa, podendo comprometer métricas como o *time-to-market*. Para facilitar e acelerar o desenvolvimento, tipicamente recorre-se à utilização de sistemas operativos. Esta *layer* de *software* introduz uma camada de abstração capaz de fornecer um conjunto de facilidades ao desenvolvimento e de gerir os recursos de *hardware*. Porém, como a mudança de paradigma sequencial para paralelo não é trivial, a extensão dos sistemas operativos para *multicore* não se tem desenvolvido ao ritmo pretendido. Além disso, as principais soluções existentes ou seguem uma abordagem AMP (*Asymmetric Multiprocessing*) - garantir *throughput* e *real-time* à custa de um elevado *footprint* de memória - ou então SMP (*Symmetric multiprocessing*) - não introduz *overhead* de memória mas pode comprometer as características de *real-time*.

Neste sentido, a presente dissertação propõe a expansão de um *µkernel* SMP para uma nova abordagem designada HMP (*Hybrid Multiprocessing*), combinando portanto as arquiteturas AMP e SMP, e garantindo assim um compromisso entre as métricas de tempo-real e *footprint* de memória. Além disso, este será implementado numa plataforma ARM *quad-core*, e será redesenhado para garantir escalabilidade.

Palavras-chave: Multicore, *µkernel*, Hybrid Multiprocessing e ARM Quad-core.

Abstract

Multicore processors are everywhere and their use in the embedded systems domain has been increasing substantially in the last years. Nowadays, the embedded systems requirements are considerably different from what they were some years ago, changing from performing very simple and specific tasks (basic control algorithms), to performing more complex and sophisticated tasks with much higher processing requirements (GUI and Internet). This growing demand of requirements at the application level wasn't followed by singlecore processors, making the migration to multicore processors inevitable.

However, developing bare-metal applications that exploit conveniently these powerfull processors can become a complex and time consuming task and may compromise metrics, such as time-to-market. In order to simplify and accelerate the development process, operating systems are typically used. This software layer introduces an abstraction that provides a set of facilities for the development and management of hardware resources. Nonetheless, as the transition from the sequential to the parallel paradigm is not trivial, then operating systems extension for multicore has not been developed as desired. Furthermore, the existing solutions or follow an AMP (Asymmetric Multiprocessing) - assure throughput and real-time at the expense of a large memory footprint - or SMP (Symmetric Multiprocessing) - don't introduce memory overhead but can compromise the real-time features - approach.

In this context, the present dissertation proposes the expansion of an SMP *µkernel* to a new approach entitled HMP (Hybrid Multiprocessing), therefore combining both AMP and SMP architectures, and so ensure a compromise between the real-time metrics and the memory footprint. Furthermore, it shall be implemented on an ARM quad-core platform, and will be redesigned to ensure scalability.

Keywords: Multicore, *µkernel*, Hybrid Multiprocessing and ARM Quad-core.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

Neste capítulo pretende-se contextualizar a temática da presente dissertação, evidenciando-se também as motivações e os objetivos principais a serem atingidos. O capítulo termina com uma apresentação geral da organização da dissertação.

1.1 Contextualização

Vivemos numa era onde a tecnologia desempenha um papel crucial nas nossas vidas. O aumento da utilização tecnológica no quotidiano tem motivado ao longo dos anos o desenvolvimento de áreas como os sistemas embebidos, que desempenham um papel relevante na resposta às crescentes necessidades das sociedades modernas. Como consequência, os sistemas embebidos evoluíram dramaticamente a nível de complexidade para conseguirem acompanhar essa crescente exigência tecnológica.

Para acompanhar essa crescente complexidade, o desenvolvimento de sistemas embebidos enfrenta cada vez mais desafios, pois tendem a integrar requisitos que são antagónicos entre si: (i) flexibilidade, (ii) fiabilidade, (iii) segurança, (iv) conectividade, (v) tempo-real, (vi) consumo energético, (vii) dimensão e forma [6]. Como consequência, as novas gerações de dispositivos, cada vez mais, integram funcionalidades tradicionalmente associadas aos sistemas de propósito geral.

Atualmente, existem tecnologias capazes de dar resposta aos novos desafios que os sistemas embebidos se têm deparado nos últimos anos, onde se destaca o processamento paralelo por ser a única solução disponível capaz de providenciar níveis

elevados de desempenho sem um grande custo energético associado. Outra tecnologia que se tem destacado são os circuitos eletricamente programáveis (FPGA – *Field-Programmable Gate Array*), com suporte para *hardware-software co-design*, que são capazes de oferecer flexibilidade na configuração do sistema e a utilização de aceleradores por *hardware*.

Assim sendo, existe uma crescente tendência na indústria para a migração de soluções *singlecore* para *multicore*. Contudo, apesar dos processadores *multicore* já serem utilizados há vários anos nos computadores pessoais e nos supercomputadores, estes não eram muito utilizados nos sistemas embebidos estando apenas presentes em dois campos dos sistemas embebidos: *mobile* (assumindo que é qualificado como um sistema embebido) e *networking*. Mas, mesmo nestes dois campos os processadores não eram utilizados como um sistema *multicore* completo, pois em geral cada *core* era dedicado a uma tarefa específica, correndo de forma independente, com pouca ou nenhuma interação entre si. Contudo, como forma de resposta aos requisitos atuais é necessário tratar estes processadores como um sistema *multicore* completo [7]. Neste sentido, surgem os sistemas operativos com suporte a *multicore*, pois estes são capazes de abstrair a complexidade do *hardware* subjacente e fornecer um conjunto de ferramentas facilitando e acelerando o desenvolvimento de aplicações, reduzindo a pressão no *time-to-market* e os desafios introduzidos pelos processadores *multicore*.

1.2 Motivações e Objetivos

Hoje em dia vivemos cada vez mais dependentes da tecnologia. Seja a nível industrial ou doméstico, os sistemas eletrónicos e informáticos desempenham um papel crucial na resposta às necessidades destes. Como tal, a complexidade e requisitos computacionais no caso dos sistemas embebidos, têm vindo a aumentar gradualmente ao longo dos anos, passando de simples algoritmos de controlo, a aplicações com interfaces gráficas e serviços de rede.

Esta exigência crescente de requisitos a nível aplicacional não foi acompanhada pelos processadores *singlecore*, devido ao inaceitável consumo energético provocado pelo aumento da frequência de relógio. Tornou-se assim inevitável a migração para os processadores *multicore* [8], sendo estes capazes de oferecer um baixo consumo energético e simultaneamente melhor *performance/throughput*. Contudo, desenvolver aplicações *bare-metal* que tirem proveito das potencialidades destes

processadores pode tornar-se uma tarefa complexa e morosa, podendo comprometer métricas como o *time-to-market* e qualidade final do produto. Para facilitar e acelerar a tarefa de desenvolvimento, tipicamente recorre-se à utilização de sistemas operativos. Esta *layer* de *software* introduz uma camada de abstração capaz de fornecer um conjunto de facilidades ao desenvolvimento (e.g. *multitasking*) e de gerir os recursos de *hardware* [9]. No entanto, a mudança de paradigma sequencial para paralelo não é trivial, pois a suposição que as instruções são executadas de forma sequencial já não é verdadeira, passando a existir verdadeira concorrência onde mais do que uma instrução pode ser executada em simultâneo. A concorrência acarreta novos desafios que antes não era necessário considerar, o que tem levado, a que a extensão dos sistemas operativos existentes para suporte *multicore* não se desenvolva ao ritmo pretendido.

Atualmente existem três soluções para explorar as capacidades dos processadores *multicore*. Uma das abordagens é o AMP (*Asymmetric Multiprocessing*), onde existe um sistema operativo por unidade de processamento, iguais ou diferentes, permitindo assim o cumprimento de requisitos de *real-time* [10], mas apresentando um elevado *footprint* de memória. Em alternativa, há também a abordagem SMP (*Symmetric Multiprocessing*), que devido a apenas existir uma cópia do sistema operativo, partilhado por todos os *cores*, apresenta um reduzido *overhead* de memória mas pode comprometer as características de *real-time*. Por último, existe ainda o conceito BMP (*Bound Multiprocessing*), que procura explorar o melhor das duas arquiteturas supramencionadas, no entanto as soluções existentes apenas se baseiam no conceito SMP mas possibilitando a atribuição de tarefas a *cores* específicos (e.g. *Neutrino RTOS*) [11].

Assim sendo, a presente dissertação propõe, como objetivo principal, o desenvolvimento de um *microkernel* HMP para uma plataforma ARM *Quad-core*. Assim, este trabalho tem como principais objetivos:

- Expansão do ARM *microkernel* SMP para uma plataforma ARM *quad-core*, garantindo a escalabilidade do sistema com um conjunto reduzido de alterações.

- Reestruturar o *microkernel* para uma configuração AMP, desenvolvendo um protótipo com pelo menos quatro instâncias e a implementação de um mecanismo de comunicação baseado no MCAPI (*Multicore Communications* API).

- Desenvolvimento de um *microkernel* HMP, combinando as duas versões desenvolvidas nos passos anteriores. Tirar partido das caraterísticas vantajosas de ambos, com o propósito de oferecer uma boa relação entre o *footprint* de memória, as métricas de *real-time*, *throughput* e a escalabilidade do sistema.
- Avaliação e comparação entre as três arquiteturas (através de *benchmarks*, *microbenchmarks*, medição de latências, etc).

1.3 Organização da Dissertação

No primeiro capítulo é feita a introdução e contextualização do trabalho. Além disso, é também apresentada a motivação da dissertação, e os objetivos a concretizar.

O capítulo 2 apresenta os fundamentos teóricos dos conceitos abordados na dissertação. Inicialmente são descritas as tendências atuais na área dos sistemas embebidos e os novos desafios que estas implicam. Seguidamente, são abordados os sistemas operativos e os seus mecanismos principais. Depois, segue-se a temática dos sistemas *multicore*, onde são apresentadas as principais abordagens de multi-processamento utilizadas pelos sistemas operativos. Além disso, no final do capítulo são, apresentados exemplos de sistemas operativos embebidos com suporte ao multi-processamento.

O capítulo 3 descreve primeiramente a arquitetura ARMv7-A. Na secção seguinte é apresentado e analisado o ARM *microkernel* SMP, utilizado como ponto de partida para a implementação do sistema final. Por fim, é apresentado o ambiente de desenvolvimento, constituído por todas as ferramentas e plataformas de desenvolvimento utilizadas durante a realização da dissertação.

No capítulo 4 são descritas todas as etapas do desenvolvimento do sistema. Assim sendo, descreve todo o trabalho desenvolvido, que inclui o *porting* inicial do ARM *microkernel* SMP para as plataformas de desenvolvimento utilizadas, o *refactoring* deste para a arquitetura AMP, e a combinação das versões SMP e AMP numa nova abordagem o HMP. Além disso, também são descritas as APIs de suporte, desenvolvidas para as arquitetura AMP e HMP. No final é descrita a expansão da arquitetura HMP para uma versão heterogénea.

No capítulo 5 são apresentados os resultados experimentais dos testes realizados.

É exposta a avaliação das diferentes versões do ARM *microkernel* desenvolvidas, relativamente aos requisitos de memória e ao desempenho. Além disso, são também apresentados os testes realizados, para as APIs de sincronização e comunicação entre sistemas operativos.

O documento termina no capítulo 6, onde são descritas as conclusões extraídas do trabalho desenvolvido, assim como algumas sugestões para o desenvolvimento futuro.

Capítulo 2

Estado da Arte

A presente dissertação insere-se na temática dos sistemas operativos embebidos com suporte a processadores *multicore*. Desta forma, torna-se fundamental neste segundo capítulo abordar as três principais temáticas inerentes à presente dissertação. Na secção inicial 2.1 a temática dos sistemas embebidos é introduzida. Seguidamente, na secção 2.2 é apresentada uma definição geral sobre sistemas operativos, bem como a distinção entre as principais arquiteturas existentes. Além disso, ainda na secção 2.2 são introduzidos os sistemas operativos embebidos, bem como os seus principais componentes. Na secção subsequente 2.3 são contextualizados os processadores *multicore* no mundo dos sistemas embebidos e apresentado um conjunto de abordagens e estratégias seguidas no desenvolvimento de sistemas operativos para plataformas *multicore*. Por fim, na secção 2.4 são apresentados exemplos de sistemas operativos existentes com suporte *multicore*.

2.1 Sistemas Embebidos

Tradicionalmente, os sistemas embebidos caracterizavam-se por serem sistemas que desempenhavam tarefas muito específicas de complexidade reduzida, como simples algoritmos de controlo, sendo limitados por restrições do *hardware* disponível [12].

Contudo, nos dias de hoje como resposta às necessidades da sociedade moderna, o prisma dos sistemas embebidos estendeu-se para sistemas de maior complexidade, levando estes a aproximarem-se, em nível de complexidade e funcionalidades, dos tradicionais computadores pessoais [12]. Esta evolução dos sistemas embebidos

tornou a definição tradicional destes obsoleta, não existindo ainda uma nova definição globalmente aceite [7].

No entanto, apesar da aproximação dos sistemas embebidos aos sistemas de propósito geral existem caraterísticas que se mantiveram exclusivas a estes. Assim sendo, é possível descrever os sistemas embebidos com base nas suas caraterísticas [13]:

- *São mais limitados a nível do hardware e/ou software quando comparados com os computadores pessoais.* Esta afirmação mantém-se verdadeira para uma grande porção dos sistemas embebidos apesar da aproximação destes aos computadores pessoais. Pois as restrições de *hardware*, nomeadamente ao nível do poder de processamento, reduzido consumo energético e pouca memória, bem como as limitações de *software* como o reduzido número de aplicações e sistemas operativos com recursos limitados, ainda se verificam para muitos destes sistemas.

- *São desenvolvidos para desempenhar tarefas específicas com requisitos pré-definidos.* Tradicionalmente, os sistemas embebidos eram desenvolvidos para desempenhar apenas uma tarefa específica. Contudo, hoje em dia existem dispositivos como as novas televisões digitais, que apesar de serem desenvolvidas com um propósito específico, também oferecem uma grande quantidade de aplicações não relacionadas com a sua função primária.

- *Apresentam maiores requisitos de qualidade, fiabilidade e disponibilidade.* Os sistemas embebidos são muitas vezes utilizados em cenários e missões onde vidas humanas se encontram em risco, onde existem requisitos muito apertados de qualidade, fiabilidade e disponibilidade.

2.2 Sistemas Operativos

Um sistema operativo tem como principal propósito abstrair a complexidade do *hardware* subjacente e fornecer um conjunto de recursos ao utilizador para que este possa desenvolver as suas aplicações de forma mais simples, rápida e eficaz [9]. Toda a complexidade da gestão de recursos, escalonamento de tarefas, gestão de ficheiros entre outros, é da responsabilidade do sistema operativo.

2.2.1 Monolítico vs Microkernel

O *kernel* é o componente principal de um sistema operativo, contendo as principais funcionalidades. Os sistemas operativos variam nos componentes que fazem parte do *kernel*, existindo duas principais arquiteturas que os caracterizam como monolíticos ou *microkernel* [13], apresentadas na figura 2.1.

Num sistema operativo monolítico, todos os serviços básicos do sistema operativo fazem parte do *kernel*, executam no espaço do *kernel*, e apenas as aplicações correm no espaço do utilizador. Como todos os serviços básicos fazem parte do *kernel*, isso introduz alguns inconvenientes como a dificuldade na gestão de funcionalidades para ser integrado em diferentes plataformas, modificação e na realização de *debug*.

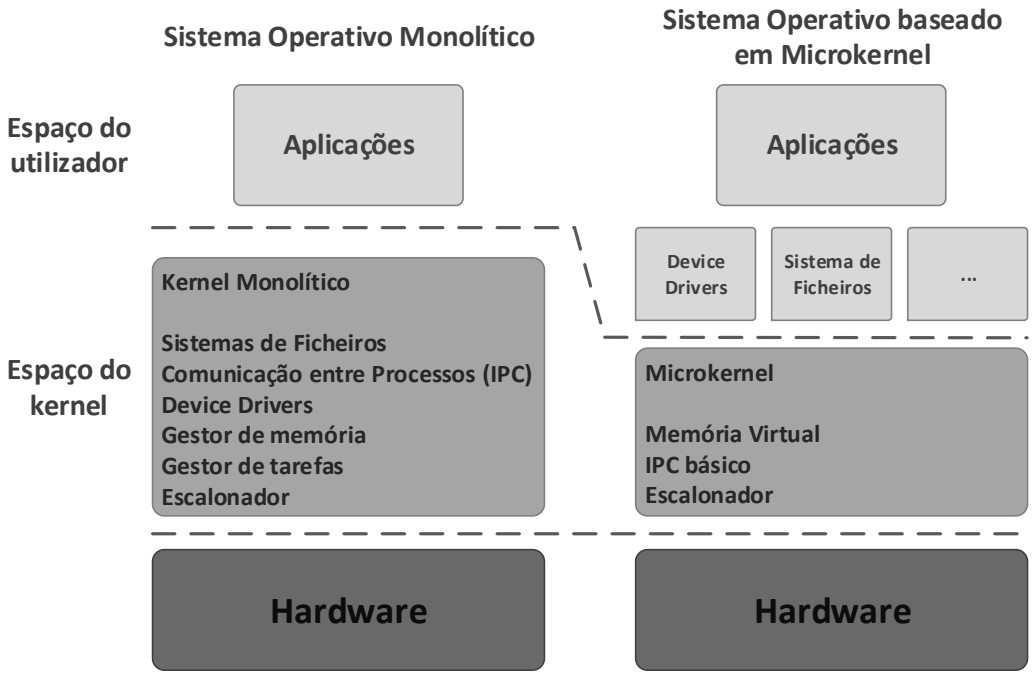

Figura 2.1: Modelos arquiteturais de um sistema operativo: monolítico e *microkernel*

Um sistema operativo que segue a arquitetura de *microkernel*, não apresenta as limitações supramencionadas, características da arquitetura monolítica. Na arquitetura *microkernel*, apenas os serviços básicos de comunicação entre processos, gestor de tarefas e gestão de memória fazem parte do *kernel*, e processos como o gestor de ficheiros e *device drivers* correm no espaço do utilizador. Esta divisão permite que componentes adicionais possam ser adicionados de forma dinâmica

facilitando assim, a realização de *debug* e tornando o sistema operativo mais escalável e seguro. Contudo, como os processos não correm no espaço do *kernel* torna-se necessário a realização de comutações de contexto, para estes entrarem e saírem do modo privilegiado [14], levando a que os sistemas operativos baseados em *microkernel* tenham um menor desempenho comparativamente aos monolíticos.

2.2.2 Sistemas Operativos Embebidos

Tradicionalmente, os sistemas operativos embebidos eram desenvolvidos com um propósito específico, pois historicamente os sistemas embebidos eram simples, possuíam restrições temporais e operavam em memória limitada [15]. Contudo, a evolução dos sistemas embebidos, nomeadamente a nível dos componentes de *hardware* (e.g. memória virtual) que estes passaram a integrar, levou a que esta distinção mudasse ao longo do tempo.

Os sistemas embebidos geralmente usam uma arquitetura *microkernel*, pois esta vai de encontro às necessidades dos sistemas embebidos, como a escassez de recursos, fácil customização e escalabilidade do sistema. Para além disso, o design de um sistema operativo embebido é fortemente afetado pelas restrições de *hardware* – a variedade de recursos e restrições de hardware que diferentes plataformas possuem obrigam o desenvolvedor a ter em consideração o uso eficiente dos recursos disponíveis – e requisitos aplicacionais – as características da aplicação para o qual o sistema operativo embebido é desenvolvido, como caraterísticas de tempo-real, podem ter de ser tidas em conta a quando o desenvolvimento de serviços deste, como o escalonador e o gestor de tarefas [16].

Gestor de Tarefas

O *multitasking* permite que o trabalho efetuado por uma aplicação seja dividido por várias tarefas [17]. Uma tarefa é um programa que executa de forma independente como se possuísse o CPU apenas para si, e pode aceder a variáveis e estruturas de dados que podem ser apenas suas ou partilhadas com outras tarefas. Um sistema operativo com suporte *multitasking* necessita de mecanismos adicionais para gerir e sincronizar as tarefas que existem em simultâneo. Isto porque, apenas uma tarefa pode estar em execução em cada instante de tempo (em sistemas *singlecore*), havendo assim a necessidade do sistema operativo alocar memória para cada tarefa,

especificar a quantidade de tempo que cada tarefa pode usar o CPU e gerir a comutação entre as diferentes tarefas.

Escalonador

Geralmente, aplicações embebidas são compostas por várias tarefas, portanto o sistema operativo tem de garantir uma distribuição dos recursos e tempo de processamento por todas as tarefas constituintes do sistema. A responsabilidade desta distribuição recai sobre o escalonador. O escalonador é a parte do sistema operativo responsável por determinar o estado (*ready*, *running*, ou *blocked*) em que cada tarefa se encontra.

Existem vários algoritmos implementados para sistemas operativos embebidos com as suas inerentes vantagens e desvantagens. Os fatores chave que afetam a efetividade e desempenho dos algoritmos de escalonamento são: (i) o tempo de resposta, (ii) tempo que demora uma tarefa a finalizar a sua execução e o (iii) *overhead* introduzido [13]. Os algoritmos de escalonamento nos sistemas embebidos seguem em geral duas arquiteturas: *non-preemptive* – é atribuído o controlo do CPU às tarefas até estas terminarem a sua execução (e.g. *run-to-completion*) – e *preemptive* – o escalonador pode interromper uma tarefa antes do fim desta e atribuir o processador a outra tarefa (e.g. *Round-Robin*).

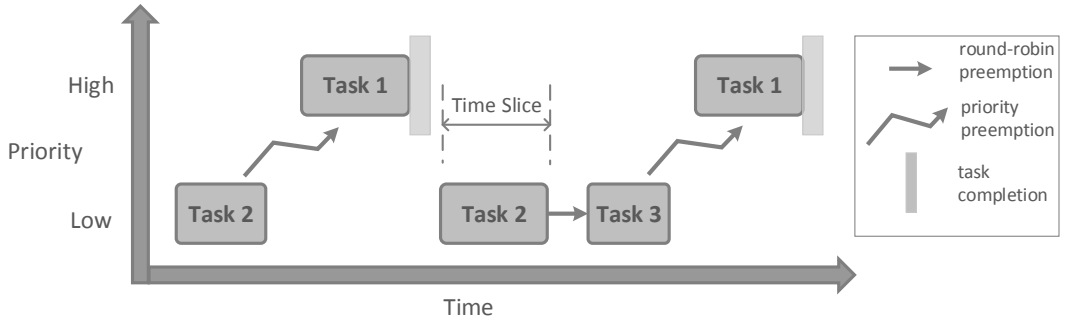

Figura 2.2: *Priority Preemptive Scheduler* com política de desempate *Round-Robin*

Nos sistemas embebidos o algoritmo de escalonamento mais usado é o *preemptive*, principalmente quando existem requisitos de tempo-real. Este algoritmo, como forma de responder à necessidade de atribuir diferentes importâncias a diferentes tarefas dentro de um sistema, também pode seguir uma política de escalonamento baseada em prioridades. Estes escalonadores denominam-se de *priority preemptive scheduler*, e permitem a atribuição de prioridades às tarefas e selecionam sempre

a tarefa ativa de maior prioridade para ser executada. Contudo, como forma de distribuir igualmente o tempo de processamento pelas tarefas de prioridade igual utiliza-se uma política de desempate *Round-Robin*. Esta política consiste, na atribuição de um *time-slice* às tarefas e sempre que este expira a tarefa é interrompida e o processamento é passado a outra tarefa de prioridade igual, como se pode constatar na figura 2.2.

Comutação de Contexto

Todas as tarefas possuem o seu próprio contexto, que consiste no estado dos registos do CPU requeridos para o correto funcionamento da tarefa [18]. As comutações de contexto ocorrem sempre que o escalonador seleciona uma nova tarefa para ser executada. Sempre que uma tarefa é criada o *kernel* também cria e mantém uma *task control block* (TCB) associada à tarefa, utilizada para manter os dados específicos da tarefa como o contexto associado a esta. Quando uma comutação de contexto ocorre, primeiro é necessário guardar o contexto da tarefa que se encontrava em execução na sua TCB e depois restaurar o contexto guardado na TCB da próxima tarefa a ser executada.

Gestor Temporização

É essencial para um sistema operativo ter conhecimento da unidade tempo, para dar suporte aos serviços temporais disponibilizados às aplicações e especialmente para implementar o relógio (*tick*) [17] do sistema operativo. Como tal, os processadores disponibilizam temporizadores que geram interrupções periódicas, usadas pelo sistema operativo para implementar os serviços temporais.

Gestor Memória

Com múltiplas tarefas a partilharem o mesmo espaço de memória, o sistema operativo necessita de mecanismos de segurança para proteger o código das tarefas, para que a atividade da tarefa que se encontra em execução não corrompa a operação às outras tarefas [15]. Para além disso, o código do sistema operativo também reside no mesmo espaço de memória das tarefas que gere, logo este precisa de proteger o seu próprio código e o das tarefas que gere. O componente do sistema operativo

responsável por esta gestão é o gestor de memória, mas para além disso também é responsável por [13]:

- Mapear entre a memória física e virtual;
- Alocar memória para as tarefas do sistema;
- Alocação dinâmica de memória;
- Assegurar a coerência da *cache* (para sistemas com *cache*);
- Assegurar a proteção da memória;

Tipicamente, uma das formas utilizadas pelos sistemas operativos para a proteção do código do sistema operativo das restantes tarefas é a divisão do espaço de memória em dois espaços de memória distintos o espaço do *kernel*, onde se encontra o código referente ao *kernel* do sistema operativo, e o espaço do utilizador, onde se encontra o código das tarefas. Esta divisão é conseguida com recurso a componentes de *hardware* como a MMU (*Memory Managment Unit*), responsável pelo mapeamento entre memória virtual e física, e restringir os acessos à memória. Para além disso, a MMU permite que cada tarefa corra de forma independente no seu espaço privado de memória, impedindo estas de aceder a áreas de memória que não as suas.

Sincronização

Num sistema embebido as tarefas que o constituem tipicamente têm de partilhar os mesmos recursos de *hardware* e *software*. A partilha de dados e de recursos pode facilitar e simplificar a troca de informação entre as tarefas, mas para garantir o correto funcionamento de todas as tarefas é necessário assegurar os acessos exclusivos para evitar a corrupção de dados partilhados. Por esta razão os sistemas operativos embebidos fornecem mecanismos que permitem que as tarefas sincronizem o acesso simultâneo aos seus recursos.

Os sistemas operativos embebidos, geralmente implementam os algoritmos de sincronização baseados na combinação de memória partilhada, passagem de mensagens e envio de sinais. Quando diferentes tarefas utilizam dados partilhados como meio de comunicação podem surgir problemas de *race conditions*. Uma *race condition* ocorre quando um processo que estava a aceder a variáveis partilhadas é escalonado antes de completar todas as modificações, pondo em causa a inte-

gridade das variáveis. Como forma de evitar as *race conditions* nestas secções dos processos denominadas de *critical sections*, utilizam-se técnicas de exclusão múltipla como bloqueios assistidos pelo processador (e.g. desativar interrupções, desativar o escalonador e *condition variables*) e semáforos que podem ser binários, de exclusão múltipla e contadores [13].

Mensagens

Quando várias tarefas coexistem no mesmo sistema pode tornar-se necessário que estas comuniquem entre si e com os serviços de atendimento às interrupções. A transferência de informação é denominada de comunicação entre tarefas, e é implementada utilizando filas de mensagens. Tipicamente, as filas de mensagens são implementadas com algoritmos FIFO *(First-in First-out)* e permitem que as mensagens enviadas a uma tarefa fiquem guardadas até que a tarefa fique novamente ativa e as possa atender, evitando assim a perda de mensagens.

2.3 Multicore

Até há poucos anos atrás, o aumento do desempenho das aplicações era ampliada de forma previsível com o aumento da velocidade de funcionamento dos processadores *singlecore*. No entanto, a frequência de funcionamento destes processadores chegou a um limite físico. O aumento da frequência de relógio tem como consequência um aumento proporcional do consumo energético [19], provocando um elevado aquecimento dos processadores e consumos energéticos elevados. Contudo, devido à natureza dos sistemas embebidos, como recursos energéticos limitados, essas consequências são inaceitáveis. Nesse sentido, os processadores *singlecore* não foram capazes de acompanhar o crescente aumento de requisitos a nível aplicacional, tornando portanto, a migração para processadores *multicore* inevitável [8].

Os processadores *multicore* combinam no mesmo SoC (*system-on-chip*) mais do que um processador *singlecore* e podem funcionar a frequências mais reduzidas, resultando num menor consumo de energia, mas mesmo assim, atingindo um aumento geral de *performance* [19]. Para além disso, estes possibilitam que várias tarefas sejam executadas em simultâneo, em paralelo, em diferentes *cores*, podendo assim, existir verdadeira concorrência ao contrário do que acontecia nos *singlecore*

onde o *multitasking* era conseguido através de técnicas de *software*, como o *time-slicing*, dando a ilusão de várias tarefas serem executadas em simultâneo [20]. Contudo, a existência de verdadeira concorrência trás novos desafios que têm de ser lidados pelos *developers*, para que as aplicações tirem o maior proveito possível das potencialidades destes processadores.

2.3.1 Asymmetric Multiprocessing

O AMP (*Asymmetric Multiprocessing*) trata cada *core* como elementos de processamento separados [19]. Um sistema AMP, pode ser homogéneo, cada *core* corre uma cópia do mesmo sistema operativo, apresentado na figura 2.3, ou heterogéneo, cada *core* corre um sistema operativo diferente [11]. Isto implica que os processos e todas as suas *threads* apenas podem correr num dos *cores*, não havendo a possibilidade de migrar tarefas entre os *cores*, o que pode levar a uma subutilização dos *cores*.

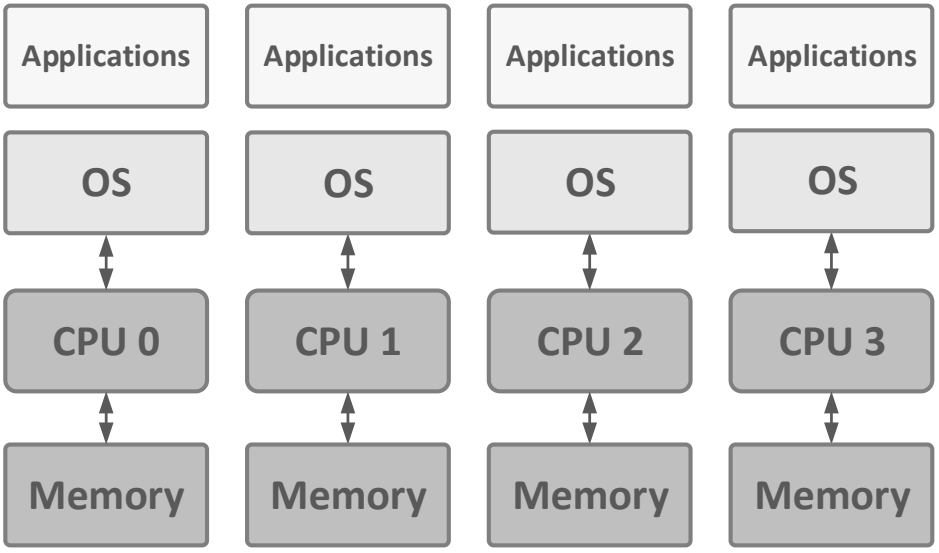

Figura 2.3: Asymmetric Multiprocessing

Contudo, o ambiente de execução providenciado é similar ao do convencional *singlecore*, o que pode facilitar a migração de código *legacy* para o ambiente *multicore*. Mas, se for necessária comunicação entre aplicações a correrem em diferentes *cores*, será necessário implementar de forma explícita a comunicação nas aplicações [19].

Algumas das razões para implementar um sistema AMP num processador *multicore*

são devido a requisitos de segurança, tempo-real, ou porque *cores* individuais são dedicados a executar tarefas específicas [10].

2.3.2 Symmetric Multiprocessing

O SMP (*Symmetric Multiprocessing*) é uma arquitetura de *software* que procura explorar as potencialidades dos processadores *multicore*. Num sistema SMP, apenas existe uma única cópia do sistema operativo, que controla todos os recursos de *hardware*, a memória é comum a todos os *cores*, e as tarefas podem ser migradas dinamicamente entre todos os *cores* [10], como se pode observar na figura 2.4. Assim sendo, é possível atingir uma melhor distribuição das tarefas pelos *cores* e oferecer uma gestão transparente dos recursos. Até certo ponto faz parecer o sistema *multicore* com um *singlecore*. Como apenas existe uma única cópia do sistema operativo, todas as *Intercore* IPC (*inter-process communication*) são locais, reduzindo assim, o *footprint* de memória e aumentando dramaticamente o desempenho, pois não há a necessidade da existência de um pesado protocolo de comunicação para implementar a comunicação entre aplicações a correrem em *cores* diferentes [11].

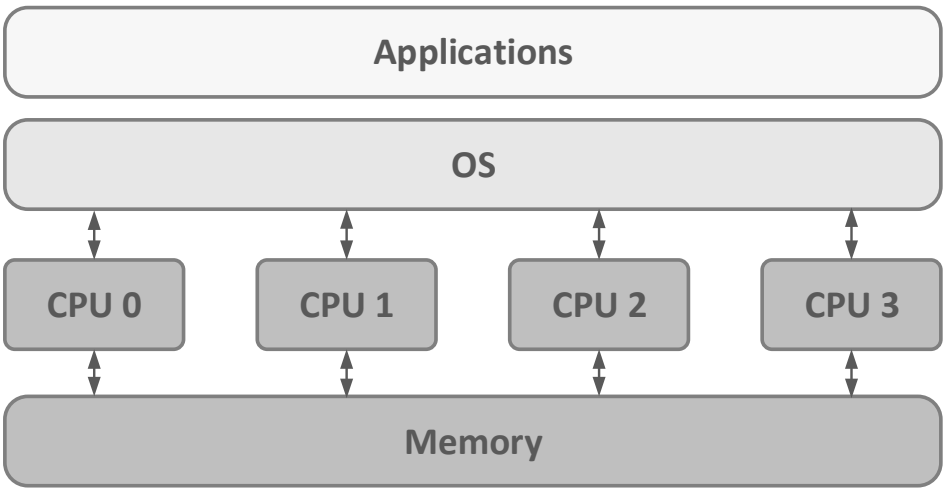

Figura 2.4: Symmetric Multiprocessing

Contudo, pode ser necessário um significativo desenvolvimento de *software* para migrar, de forma eficiente, aplicações *legacy* para um sistema *multicore* SMP, pois o suporte ao paralelismo pode não ter sido considerado explicitamente no *software* original [19]. Além disso, a migração dinâmica de tarefas pode nem sempre significar uma utilização mais eficiente dos *cores*, devido ao *cache trashing* que pode

ocorrer a quando da migração das tarefas, onde a *cache* do novo *core* tem de ser preenchida com dados relevantes presentes na *cache* do antigo *core* [11].

2.3.3 Bound Multiprocessing

O BMP (*Bound Multiprocessing*) surge como solução a alguns dos problemas apresentados pela arquitetura SMP [21]. Um sistema BMP oferece os benefícios da gestão transparente de recursos do SMP e o controlo do escalonamento de tarefas do AMP [7], pois este dá ao designer a possibilidade de atribuir qualquer aplicação (e as suas respetivas *threads*) a um *core* específico. Esta capacidade de atribuição de tarefas a *cores* torna mais fácil a migração de código *singlecore* para o ambiente *multicore* e ajuda a eliminar o *chache trashing*, que por vezes pode reduzir o desempenho dos sistemas SMP.

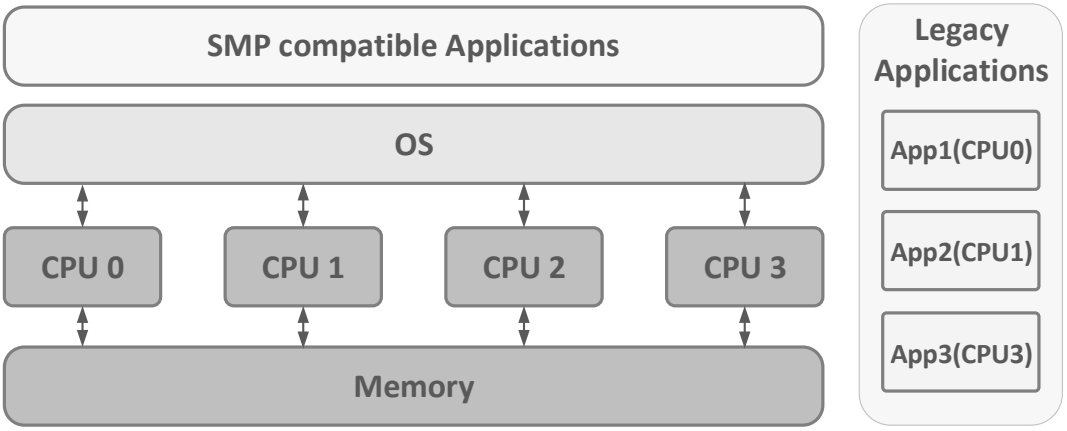

Figura 2.5: Bound Multiprocessing

Tal como a arquitetura SMP, apenas existe uma única cópia do sistema operativo que mantém uma visão geral e unificada dos recursos do processador, permitindo que estes sejam dinamicamente alocados entre as aplicações. Assim sendo, um sistema BMP permite a coexistência de aplicações *legacy* com aplicações desenvolvidas para tirar o máximo proveito dos processadores *multicore* [11], como pode ser observado na figura 2.5.

2.4 Sistemas Operativos Multicore

Atualmente, já existem soluções de sistemas operativos embebidos com suporte *multicore* disponíveis no mercado, baseados nas abordagens previamente apresentadas na secção 2.3. Alguns exemplos destes sistemas operativos são apresentados a seguir.

2.4.1 Nucleus

Desenvolvido pela Mentor Graphics, o Nucleus é um sistema operativo de tempo-real baseado em *microkernel*, desenhado para ser altamente escalável e fiável. O sistema operativo Nucleus é o primeiro sistema operativo de tempo-real na indústria a oferecer um *power-aware kernel*, disponibilizando uma *framework* para gestão de energia integrada no sistema operativo. Para além disso, o Nucleus também oferece serviços de *networking*, interface gráfica, sistema de ficheiros, conetividade e segurança, como se pode verificar na figura 2.6. Também oferece suporte a várias plataformas de hardware como ARM, MicroBlaze, MIPS, Power e Nios II [1].

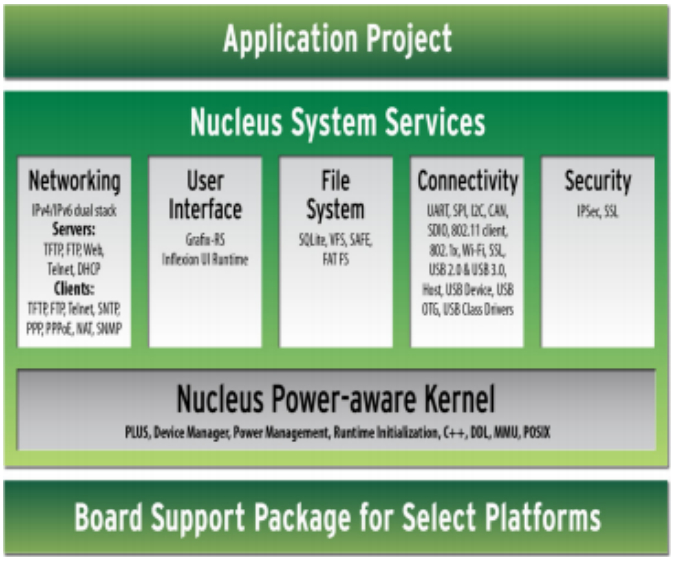

Figura 2.6: Arquitetura do Nucleus [1]

O Nucelus, atualmente, também possui suporte para plataformas *multicore*, como o *Asymmetric Multiprocessing* e o *Symmetric Multiprocessing* com suporte a *core affinity*, além disso, também possui um escalonamento *multicore* preemptivo e determinístico [22].

2.4.2 Neutrino

O Neutrino, desenvolvido pela QNX Software Systems, é provavelmente o líder no mercado de sistemas operativos baseados em *microkernel*, especialmente na área dos sistemas embebidos [23]. A arquitetura *microkernel* presente no Neutrino, apresentada na figura 2.7, possibilita que este seja altamente escalável tornando-o ideal para sistemas embebidos de tempo-real, pois este pode ser reduzido a um tamanho consideravelmente pequeno e continuar a disponibilizar *multitasking*, *threads*, escalonamento preemptivo baseado em prioridades, e comutação de contexto rápida. Contudo, este também pode ser dotado de funcionalidades como gestor de ficheiros, interface gráfica, gestor de rede nativo entre outras, podendo assim ser usado em sistemas que excedem os requisitos dos sistemas embebidos. Para além do que foi previamente mencionado, o Neutrino também utiliza a API *standard* da POSIX e suporta várias famílias de processadores, incluindo x86, ARM, XScale, PowerPC, MIPS e SH-4 [2].

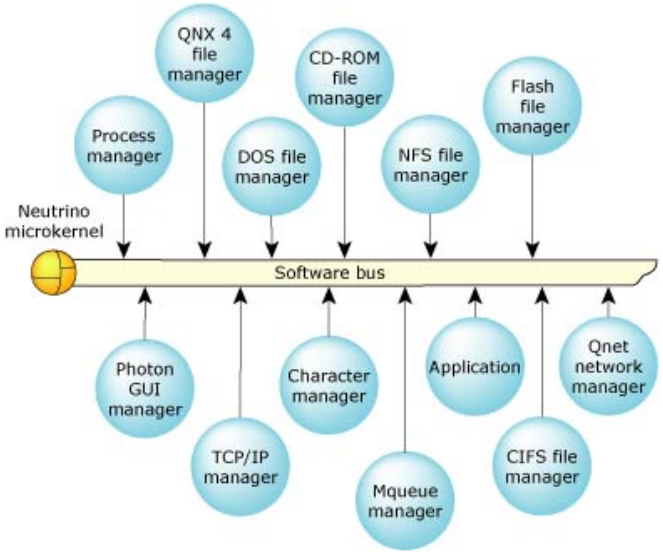

Figura 2.7: Arquitetura do Neutrino [2]

O sistema operativo da QNX foi também um dos primeiros sistemas operativos embebidos a ter suporte *multicore*, e atualmente suporta *Asymmetric Multiprocessing*, *Symmetric Multiprocessing* e *Bound Multiprocessing*, sendo este último uma inovação introduzida pela QNX para facilitar a migração de código *legacy* para plataformas *multicore*, como visto na secção 2.3.

2.4.3 ARM microkernel SMP

Com o propósito de demonstração e investigação, a ARM implementou um ambiente de execução *multi-threading* SMP leve em *bare-metal* e uma biblioteca de sincronização. Este foi implementado para correr nos processadores ARM11 *MPCore* e Cortex A9 *MPCore* e implementa um escalonador *preemptive* simples e oferece uma porção da API *standard* da POSIX [24].

Capítulo 3

Especificação do sistema

No capítulo anterior foram apresentados os conceitos essenciais ao enquadramento da temática da dissertação. No presente capítulo, serão analisados os diferentes componentes do sistema implementado, bem como as ferramentas de desenvolvimento utilizadas. Assim, na secção 3.1 é descrita a arquitetura do processador no qual o sistema foi implementado, bem como o suporte ao processamento paralelo que este disponibiliza. Posteriormente, na secção 3.2, é realizada uma análise da implementação do sistema operativo embebido ARM *Microkernel* SMP, com o intuito de compreender o funcionamento deste, tendo como principal foco o suporte ao processamento paralelo. Finalmente, na secção 3.3 são analisadas as plataformas (Xilinx Zynq-7000 e VE) e *toolchains* de desenvolvimento (Xilinx ISE Design Suite e ARM FastModels) utilizadas.

3.1 Arquitetura ARM

Os processadores ARM (*Advanced RISC Machine*) baseiam-se na arquitetura RISC (*Reduced instruction Set Computer*) [15]. A filosofia RISC tem como objetivo disponibilizar um conjunto de instruções simples, capazes de serem executadas num único ciclo de relógio, a elevada frequência.

Contudo, como forma de atingir os requisitos do mercado alvo, principalmente sistemas embebidos, os processadores ARM foram especialmente desenvolvidos de forma a apresentarem um consumo energético reduzido e elevada densidade de código. Estes requisitos levam a que os processadores ARM não sigam uma arqui-

tetura RISC pura. Além disso, dentro da arquitetura ARM existem três variantes com diferentes propósitos [10]: (i) *Application profile* - destinada a processadores de elevado desempenho, (ii) *Real-Time profile* - destinada a sistemas que requerem determinismo temporal e (iii) *Microcontroller profile* - destinada a sistemas de baixo custo onde a baixa latência no processamento de interrupções é vital.

3.1.1 ARM Cortex-A9

A família de processadores Cortex-A é estruturada como um computador completo, capaz de executar sistemas operativos complexos. Estes processadores são utilizados principalmente nos telemóveis, *tablets* e computadores portáteis.

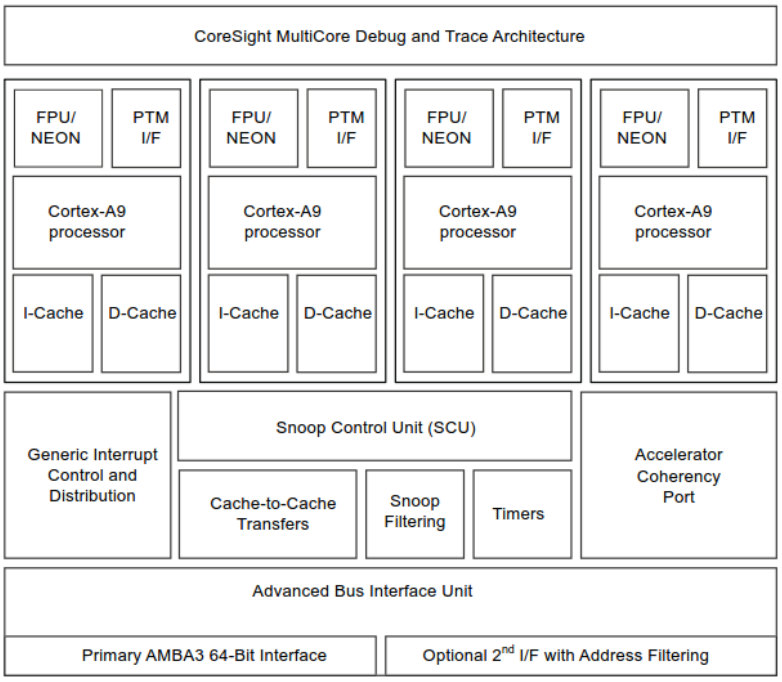

Figura 3.1: Diagrama de blocos do processador *multicore* Cortex-A9

O ARM Cortex-A9, baseado na arquitetura ARMv7-A, é um processador de elevada *performance* que apresenta um consumo eficiente de energia. Este pode ser configurado até quatro *cores* sendo assim, capaz de fornecer um maior desempenho sem um elevado custo energético associado.

O Cortex-A9 foi desenvolvido com o propósito de providenciar um elevado desempenho, utilizando para esse efeito, uma arquitetura de processamento fora de ordem *dual-issue superscalar*. Este também suporta *caches* L1 *four way associative* de 16, 32 ou 64 KB, e um controlador opcional para uma *cache* L2 até 8MB. Além disso,

também disponibiliza uma unidade de processamento de números reais, tecnologia NEON para processamento de multimédia e processamento SIMD.

3.1.2 Modos de execução

Nos processadores baseados na arquitetura ARMv7-A, são disponibilizados nove modos de execução que determinam quais são os recursos do sistema que podem ser acedidos. Os modos de execução podem ser privilegiados ou não-privilegiados. Os modos privilegiados têm total acesso a todos os recursos do sistema e podem comutar de modo livremente. Por outro lado, nos modos não-privilegiados o acesso aos recursos do sistema é limitado e apenas podem comutar de modo com recurso a exceções. Assim sendo, os modos de execução presentes na arquitetura ARMv7-A são os seguintes [25]:

- **User Mode:** É o único modo não-privilegiado, presente na arquitetura ARMv7-A, onde a maioria das aplicações são executadas. Como dito anteriormente, neste modo as aplicações não têm acesso aos recursos protegidos e só poderão mudar o modo de execução desencadeando exceções.
- **FIQ Mode:** Acedido sempre que é desencadeada uma interrupção FIQ.
- **IRQ Mode:** Acedido sempre que é desencadeada uma interrupção IRQ.
- **Supervisor Mode:** Acedido sempre que o processador é reiniciado ou quando uma instrução SVC (*Superviser Call*) é executada.
- **Monitor Mode:** Faz parte das extensões de segurança introduzidas pela *TrustZone*. É sempre executado em modo seguro e permite a comutação entre o mundo seguro e não seguro. Pode ser acedido com a execução da instrução SMC (*Secure Monitor Call*) ou com o desencadeamento de exceções configuradas para serem atendidas no modo Monitor.
- **Abort Mode:** Acedido sempre que uma exceção *prefetch abort* ou *data abort* é desencadeada. Por outras palavras, é acedido sempre que o processador falha um acesso à memória.
- **Hyp Mode:** Faz parte das extensões de suporte à virtualização. Pode ser acedido com a execução da instrução HVC (*Hypervisor Call*).
- **Undef Mode:** Acedido sempre que o processador encontra uma instrução

não definida ou não suportada.

- **System Mode:** É uma versão especial do modo User que tem acesso total a todos os recursos do sistema. Este modo apenas pode ser acedido através da escrita no registo CPSR (*Current Program Status Register*).

3.1.3 Registos

A arquitetura da ARM disponibiliza dezasseis registos de propósito geral, onde estão incluídos o *Stack Pointer* (R13 - SP), *Link Register* (R14 - LR) e o *Program Counter* (R15 - PC). Além dos registos de propósito geral também é disponibilizado o *Current Program Status Register* (CPSR) e o *Saved Program Status Register* (SPRS), que guarda o CPSR do modo anterior em execução.

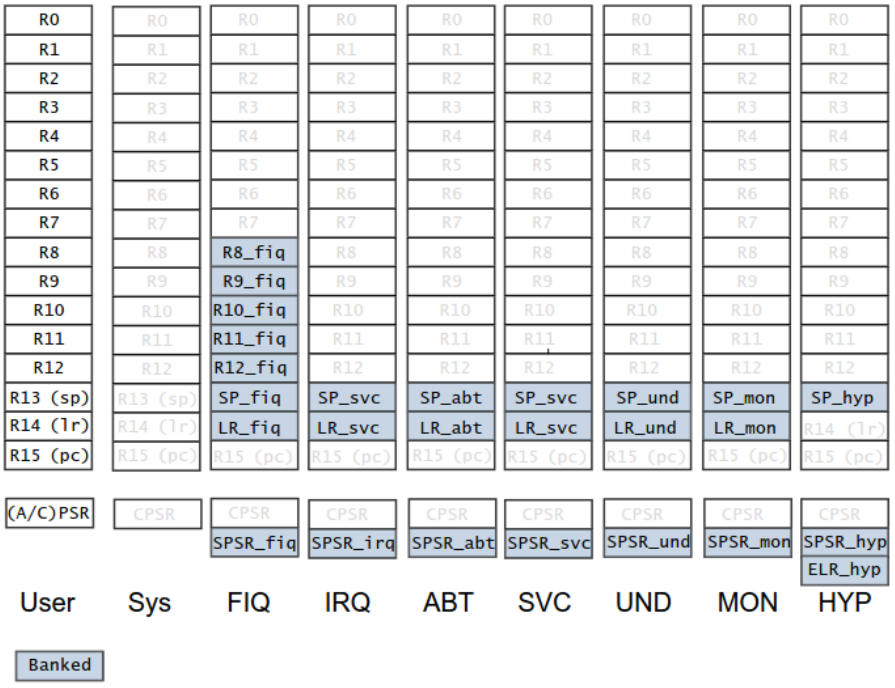

Figura 3.2: ARMv7-A *Register File*

Como se pode observar na figura 3.2, cada modo de execução possui uma versão apenas sua de certos registos. Como por exemplo, o modo *Supervisor* possui um SP, LR e SPSR apenas acessíveis no modo *Supervisor*.

3.1.4 *Generic Interrupt Controller*

O GIC (*Generic Interrupt Controller*) define quais os recursos de *hardware* que geram as interrupções, num sistema *single* ou *multicore* [26]. Como forma de possibilitar a configuração das fontes de interrupções e o seus comportamentos, o GIC disponibiliza registos mapeados em memória. Além disso, também possibilita a ativação/desativação de interrupções, atribuição de prioridades (em *hardware*) e o desencadeamento de interrupções por *software*.

A nível da arquitetura, o GIC encontra-se dividido em dois grandes blocos funcionais:

- ***Distributor*:** onde todas as fontes de interrupção no sistema estão conectadas. O *distributor* possui registos que controlam as propriedades individuais de cada interrupção. É também o *distributor*, que num sistema *multicore*, determina para qual dos *cores* uma determinada interrupção deve ser enviada.

- **CPU *Interface*:** responsável por receber as interrupções enviadas para um *core*. Existe uma CPU *Interface* por cada *core*. Esta possui registos, acessíveis por *software*, que identificam e controlam o estado das interrupções recebidas.

Cada interrupção é identificado por um ID único, definido pelo design do sistema. As interrupções também podem ser categorizadas de acordo com a sua natureza:

- **SGI (*Software Generated Interrupt*):** Interrupções geradas por *software*. São essencialmente utilizadas para comunicação entre *cores*. Podem ser destinadas a todos os *cores* ou a um grupo específico. Os IDs 0-15 são reservados para estas interrupções.

- **PPI (*Private Peripheral Interrupt*):** Interrupções geradas por periféricos privados de um *core*. Os IDs 16-31 são reservados para estas interrupções.

- **SPI (*Shared Peripheral Interrupt*):** Interrupções geradas por periféricos partilhados por todos os *cores* presentes no sistema. Os IDs 32-1019 são reservados para estas interrupções.

3.1.5 Modelos de memória

Os processadores atuais implementam otimizações na forma como as instruções são executadas, e como os acessos à memória são realizados [10]. Como forma de esconder a latência introduzida pelos acessos à memória são utilizadas *caches* e *write buffers*, contudo estes componentes introduzem não linearidades nos acessos à memória. A ordem de execução dos acessos à memória, por um *core*, não será necessariamente a mesma que outros dispositivos externos irão ver.

Os processadores da série Cortex-A9 da ARM implementam um modelo de memoria *weakly-ordered*. Contudo, dentro deste modelo é possível especificar regiões de memória como sendo *strongly-ordered*.

- **Memória *strongly ordered***: é garantido que as transações de memória ocorrem na mesma ordem pela qual são lançadas. Além disso, estas também são imediatamente visíveis por todos os *cores*.
- **Memória *weakly ordered***: a ordem de execução e visibilidade das transações de memória não é garantida, para todos os *cores*. Contudo, este modelo de memória oferece um maior desempenho.

Assim, o sistema de memória da ARM permite as seguintes configurações [24]:

- ***Normal***: memória *weakly ordered*, apenas usada por um *core*.
- ***Normal Shared***: memória *weakly ordered* utilizada por mais do que um *core*. Coerência da *cache* L1 é mantida pela SCU (*Snoop Control Unit*) quando o core se encontra no domínio SMP 3.1.6.
- **Memória *strongly ordered***: Todos os acessos à memória ocorrem na ordem com que são descritos no *software*. Todos os acessos são considerados partilhados.
- ***Device***: memória *strongly ordered* normalmente utilizada para os registos dos periféricos.

Barreiras de memória

Como vimos anteriormente, as otimizações como *caches* e *write buffers* podem levar à alteração da ordem em que os acessos à memória são efetuados. Nesse

sentido, arquitetura da ARM especifica instruções que funcionam como barreiras de memória [27]:

- ***Data Synchronization Barrier* (DSB)**: força o *core* a esperar que todos os acessos a dados sejam efetuados antes que uma nova instrução seja executada.
- ***Data Memory Barrier* (BMB)**: garante que todos os acessos à memória, pela mesma ordem que são descritos no *sotfware*, sejam visíveis no sistema, antes que um novo acesso explícito à memória posterior à barreira apareça.
- ***Instruction Synchronization Barrier* (ISB)**: assegura que todas as instruções que aparecem a seguir à barreira sejam lidas da *cache* ou memória, descartando todas as instruções lidas anteriormente.

3.1.6 Processamento paralelo

Um processador *multicore* ARM pode conter entre um a quatro *cores* [10]. Cada *core* pode ser configurado, por *software*, para participar na gestão da coerência entre as *caches* de dados. Essa gestão é assegurada de forma automática, por um componente de *hardware*, denominado de *Snoop Control Unit*.

Snoop Control Unit

A SCU é responsável por manter a coerência entre as *caches* de dados de cada *core*. A manutenção da coerência é implementada utilizando um protocolo baseado no protocolo MOESI. Na figura 3.3 pode-se observar o processo de manutenção da coerência entre as *caches* de dados do CPU0 e do CPU2.

Para que a manutenção da coerência se encontre ativa, para um acesso à memoria, é necessário que se verifiquem certas condições:

- A SCU tem de se encontrar ativa;
- O *core* que executa o acesso tem de estar configurado para participar no *Inner Shareable domain* (o bit `SMP` no `CP15:ACTLR` ativo);
- A MMU estar ativa;

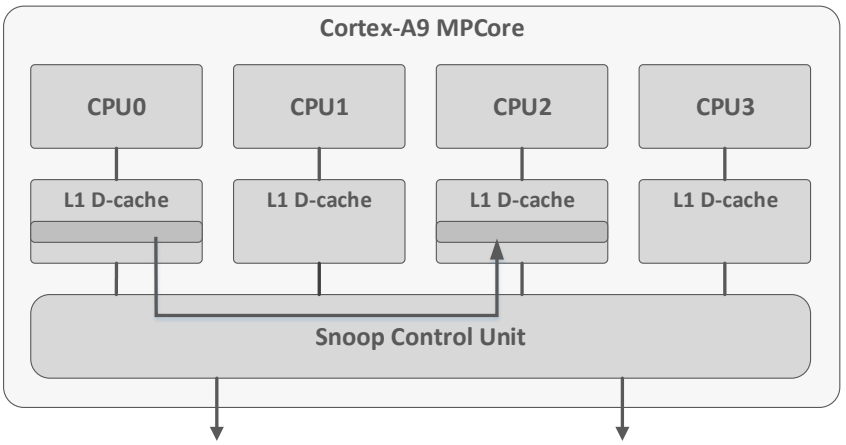

Figura 3.3: Funcionamento da SCU

- A pagina de memória a ser acedida tem de estar marcada como *normal shared*, com uma política de acesso à *cache write-back, write-allocate*.

Instruções atómicas

Num sistema *singlecore*, exclusão mútua pode ser facilmente conseguida desabilitando as interrupções dentro de secções de código críticas. Contudo, num sistema *multicore* isso já não é suficiente, pois a desativação das interrupções num *core* não impede que outros *cores* acessem a essas secções [10]. Assim sendo, é essencial que um sistema *multicore* disponibilize suporte, ao nível do *hardware*, que possibilite a implementação correta e eficiente de mecanismos de exclusão mútua entre os vários *cores* presentes no sistema.

Nesse sentido, os processadores ARM providenciam um par de instruções para ler e atualizar a memória. Estas instruções dependem da capacidade do *core*, ou do sistema de memória, em marcar os endereços de memória para serem monitorizados para acesso exclusivo, utilizando, para esse efeito, o *exclusive access monitor*.

- `LDREX` (*Load Exclusive*): lê da memória, mas também marca o endereço para ser monitorizado para acesso exclusivo.
- `STREX` (*Store Exclusive*): apenas guarda na memória se o endereço estiver marcado para acesso exclusivo. A instrução retorna 1 se o *store* não for executado, e 0 se este acontecer.
- `CLREX` (*Clear Exclusive*): remove todos os acessos exclusivos a serem moni-

torizados pelo *core*.

Existem restrições na utilização destas instruções que são relevantes ter em consideração de forma a garantir o seu correto funcionamento. Algumas das mais relevantes são apresentadas abaixo. A lista completa pode ser consultada em [27].

- O `STREX` e o `LDREX` apenas podem ser executados em memória normal, e têm efeitos diferentes dependendo se a memória se encontrada configurada como partilhada ou não.

- Cada *core* apenas pode ter um endereço marcado para ser monitorizado. O *exclusive access monitor* local, não tem de guardar o endereço de memória, por isso a arquitetura permite que o *exclusive access monitor* corresponda um `STREX` com um `LDREX` precedente, independentemente do endereço. Por isso, a instrução `CLREX` deve ser usada a quando da execução de uma comutação de contexto, para impedir que um `STREX` seja correspondido com um `LDREX` executado para um endereço diferente.

- O *global monitor* apenas suporta um acesso exclusivo, a memória partilhada, por *core*. Um `LDREX` executado por um *core*, não tem qualquer efeito no estado do *global monitor* para outro *core*.

- A possibilidade de executar as instruções, em regiões de memória definidas como de periféricos e *strongly ordered*, depende da implementação do processador.

3.2 ARM *Microkernel* SMP

Como mencionado na secção 2.4.3, o *Microkernel* SMP desenvolvido pela ARM, é constituído por um simples escalonador *preemptive* e oferece uma porção da API standard da POSIX. Este foi desenvolvido para correr nas versões *quad-core* dos processadores ARM11 *MPCore* e Cortex-A9 *MPCore*. Contudo, como este *microkernel* foi desenvolvido com o propósito de demonstração, este não se encontra otimizado e não explora a escalabilidade permitida pela tecnologia ARM *MPCore*. Seguidamente, nesta secção será analisada a implementação do ARM *Microkernel* SMP.

3.2.1 *Boot*

Num processador *multicore*, ao contrário do que acontece num processador *single-core*, é necessário inicializar mais do que uma unidade de processamento. Assim sendo, é necessário ter em consideração novos aspetos, como a partilha de recursos de hardware, para a correta inicialização do sistema operativo.

Na figura 3.4 pode-se observar os vários passos que constituem o processo de inicialização do ARM *microkernel* SMP. Nesta mesma figura, também se pode constatar que durante o processo de inicialização existe uma distinção entre os CPUs. Isto, porque um dos CPUs é responsável por coordenar o processo de inicialização de todos os CPUs, levando a que este seja denominado de CPU primário e os restantes de CPUs secundários. Para além disso, o CPU primário também é responsável por executar todas as inicializações que apenas devem ser executadas uma vez, como por exemplo, a inicialização de recursos de *hardware* partilhados.

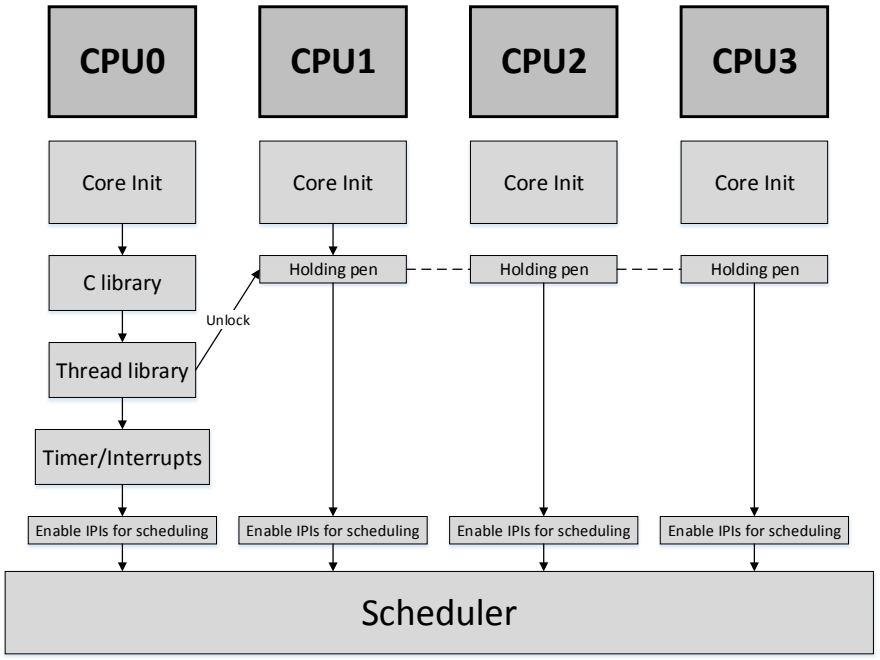

Figura 3.4: Boot

Tendo em consideração as particularidades do processo de inicialização supramencionadas, este pode ser dividido em três partes distintas:

- **Código de inicialização comum**
  O código de inicialização comum assenta principalmente na configuração de cada CPU. Assim, nesta secção de código serão inicializadas as *stacks* (pilhas)

de todos os modos de operação utilizados (SVC, IRQ e SYS), desativado o sistema de memória - *caches* e a MMU -, instalado do vetor de exceções, indicada a participação no domínio SMP, instalada a MMU *translation table* e ativadas a MMU e otimizações como o *branch prediction*.

- **Código apenas executado pelo CPU primário**
  Esta secção de código contém todas as inicializações que apenas devem ser executadas uma vez. É nesta secção que são inicializados todos os periféricos partilhados como a SCU e o GIC, e também todas as bibliotecas de suporte (biblioteca de C e das *threads*). Além das inicializações, também possui o código referente à coordenação do arranque de todos os CPUs e a invocação da função `main()`.

- **Código apenas executado pelos CPUs secundários**
  Por fim, esta secção de código possui uma função que permite que os CPUs secundários aguardem pela conclusão das inicializações realizadas apenas pelo CPU primário, e a invocação da função `cpu_idle()`, onde são inicializadas as *CPU Interfaces* de cada CPU, e é enviado um sinal ao CPU primário, sinalizando que estes se encontram prontos para executar *threads*.

Um aspeto muito importante a ter em consideração é a sincronização do arranque dos CPUs. Como acabamos de ver, o código encontra-se seccionado em três partes distintas, sendo portanto necessário coordenar a execução dessas secções, de forma a garantir a correta inicialização de todo o sistema. Nesse sentido, tal como podemos observar na figura 3.4, enquanto o CPU primário executa as suas funções exclusivas, todos os restantes CPUs aguardam, até que este os desbloqueie novamente. Apesar de não ser visível na figura 3.4, existe mais um momento onde é requerida a sincronização entre todos os CPUs, pois o escalonador apenas arranca após todos os CPUs secundários terem sinalizado, o CPU primário, que já se encontram prontos. A partir deste ponto, a distinção inicial entre os CPUs desaparece, e todos os CPUs encontram-se prontos a executar *threads*.

3.2.2 Gestão da memória

Atualmente, vários sistemas operativos recorrem a recursos de *hardware* como, a MMU, para configurarem a memória. Neste sentido, o ARM *microkernel* SMP utiliza a MMU da ARM para configurar a memória com o *layout* apresentado na figura 3.5. A *translation table* é definida estaticamente, sendo gerada utilizando

diretivas do compilador. Esta implementa um esquema de mapeamento direto onde todos os endereços virtuais correspondem aos endereços físicos.

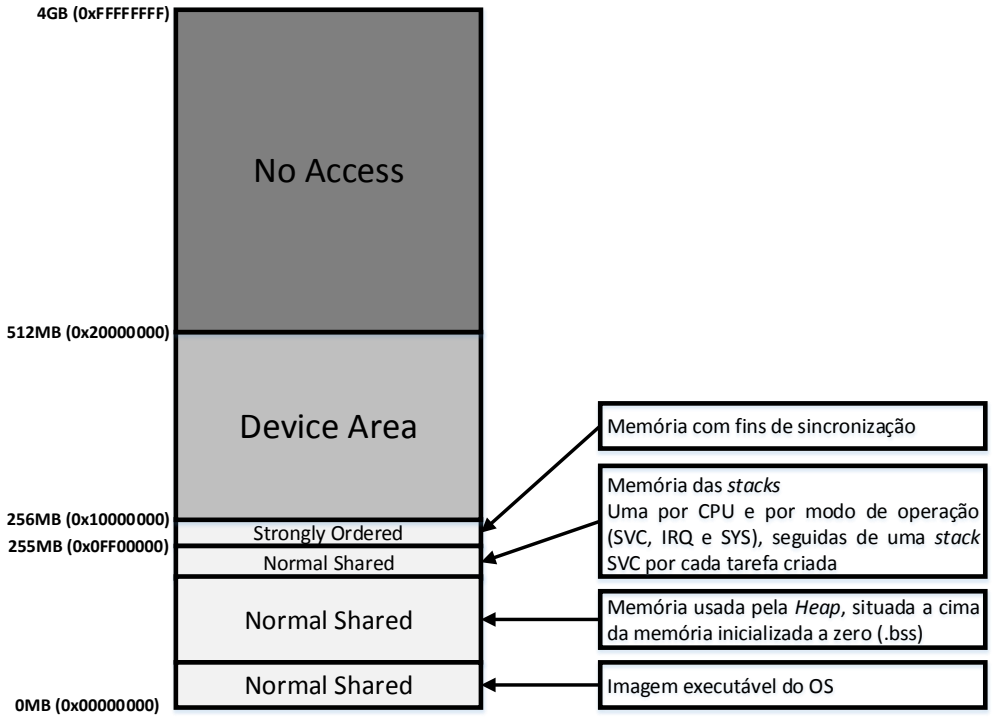

Figura 3.5: Configuração da memória

Neste caso, a MMU é essencialmente utilizada de forma a cumprir os requisitos para a ativação da SCU (secção 3.1.6), vital para manter a coerência da memória entre todos os *cores*. Assim sendo, toda a memória disponível é configurada como sendo normal *shared*, com exceção da memória referente aos periféricos, bem como um *megabyte* que é configurado como *strongly ordered*, com o propósito de ser utilizado para sincronização antes da coerência entre *caches* se encontrar ativa.

Como referido na secção 3.2.1, cada *core* inicializa as suas *stacks*, para cada modo de operação. Os *offsets* entre as *stacks*, de cada modo, são calculados a partir do identificador de cada *core*. Além das *stacks* dos modos de operação, também é gerada uma *stack* por cada *thread* criada. A organização das *stacks* pode ser observada na figura 3.6.

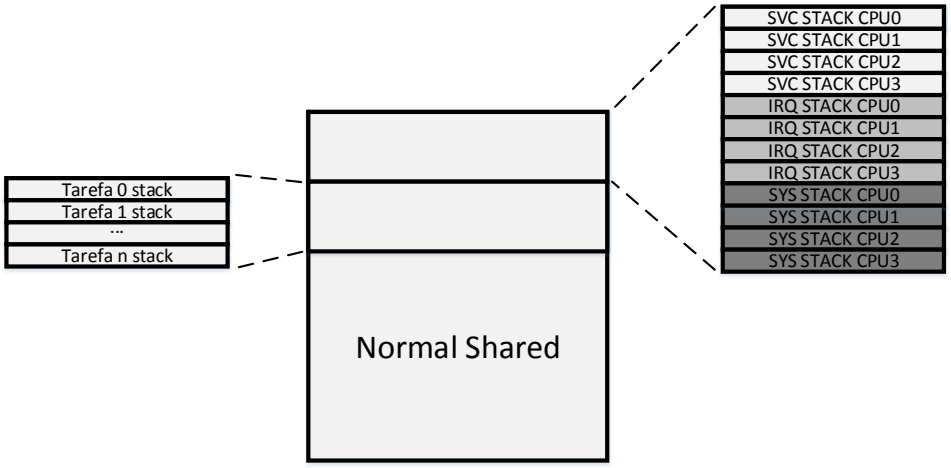

Figura 3.6: Organização das *stacks*

3.2.3 Escalonador

O escalonador é o componente do sistema operativo responsável por decidir qual a tarefa que deve estar em execução em cada instante de tempo. Existem vários algoritmos de escalonamento, dependendo da política de escalonamento que adotam (e.g. *preemtive*, *non-preemptive*, suporte a prioridades). No caso dos sistemas operativos desenvolvidos para processadores *multicore*, o escalonamento pode também suportar afinidade de tarefas, isto é, garante-se que uma determinada tarefa apenas irá ser executada num *core* específico.

No caso do ARM *microkernel* SMP foi implementado um simples escalonador *round-robin*, não otimizado. Todas as *threads* ativas encontram-se referenciadas numa estrutura de dados denominada de `PendingPCBs`. Esta estrutura de dados é maioritariamente gerida pelo escalonador, pois este utiliza esta estrutura para determinar quais são as próximas *threads* a serem executadas. Nesse sentido, o escalonador organiza a `PendingPCBs` com o *layout* apresentado na figura 3.7, dividindo assim as *threads* em três grupos:

1. ***Threads* válidas**: *threads* criadas pelo utilizador prontas a serem executadas.

2. ***Dummy threads***: *threads* criadas pelo sistema operativo com o intuito de ocupar o processador quando não há *threads* válidas prontas a serem executadas. Existe uma por cada unidade de processamento.

3. ***Threads* adormecidas**: *threads* criadas pelo utilizador que se encontram

bloqueadas ou em espera.

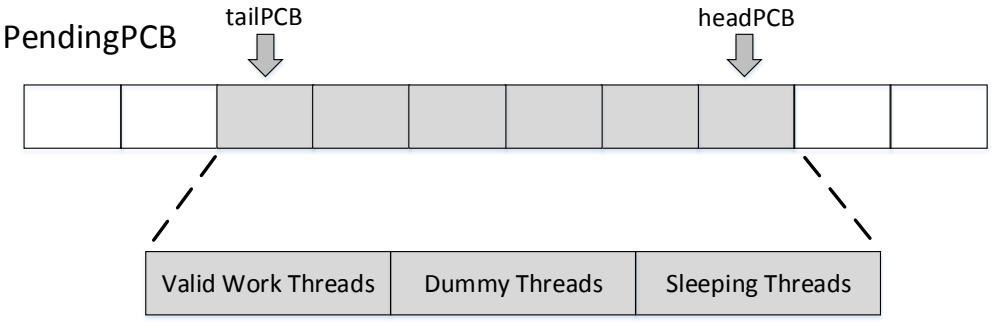

Figura 3.7: Fila de tarefas pendentes

Com esta organização da `PendingPCBs` o sistema operativo garante que sempre que existam *threads* válidas estas serão sempre executadas, e só quando nenhuma se encontra pronta a ser executada as *dummy threads* irão ocupar o processador.

Tick do escalonador

Nos escalonadores *round-robin*, é necessário ter um mecanismo que gere um *tick* periódico para implementar o *time-slice* necessário. Geralmente, o *tick* é obtido a partir do *trigger* da interrupção de um temporizador. Contudo, nos sistemas operativos *multicore* é também necessário distribuir o *tick* por todos os *cores* constituintes do sistema.

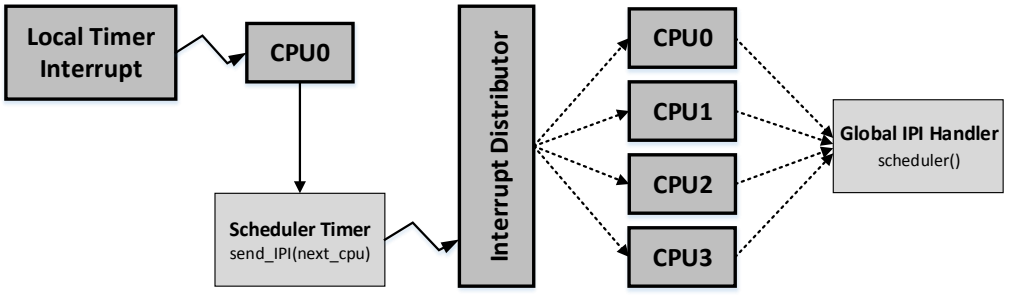

Figura 3.8: *Tick* do Escalonador

Assim sendo, no ARM *microkernel* SMP, durante a inicialização, o temporizador privado pertencente ao CPU primário é programado para gerar o *tick* a cada *time-slice*. A cada *tick* uma IPI é enviada para um dos *cores* do sistema, seguindo um esquema *round-robin*, com o propósito de desencadear a função de escalonamento,

garantindo assim que cada *core* executa a função de escalonamento a cada *time-slice*. Este mecanismo pode ser observado na figura 3.8.

3.2.4 Gestão de Tarefas

Num sistema operativo *multicore* pode-se encontrar mais do que uma tarefa em execução simultânea, pois existe verdadeiro paralelismo. No entanto, o número de tarefas que o sistema operativo gere é geralmente superior ao número de unidades de processamento disponíveis, sendo por isso necessários mecanismos que permitam a comutação entre tarefas.

Como descrito anteriormente, o escalonador é o mecanismo responsável pela seleção das tarefas que se encontram em execução a cada instante de tempo. Contudo, para que o processamento possa ser corretamente transferido de uma tarefa para outra é necessário salvaguardar a informação sobre o estado desta, designado como contexto da tarefa. Assim sendo, sempre que uma tarefa é suspendida, o estado do processador é guardado, sendo restaurado quando a tarefa é resumida. Esse estado consiste geralmente num apontador para a próxima instrução a ser executada, o conteúdo dos registos e as *flags* do processador. No caso dos processadores de arquitetura ARM, apenas os registos referentes ao modo em que as tarefas são executadas necessitam de ser preservados, logo no caso do ARM *microkernel* SMP, o estado do processador consiste nos registos do modo *supervisor* e as *flags* do processador guardadas no registo SPSR do modo IRQ.

Com o intuído de possibilitar a manutenção e organização das tarefas e respetivos contextos, o ARM *microkernel* SMP guarda as informações relevantes de cada tarefa numa estrutura de dado designada de PCB (*Process Control Block*). Dentro da PCB existe também uma referência para uma estrutura de dados responsável por armazenar informação adicionais utilizadas pelo sistema operativo na gestão das tarefas:

- **Identificador:** identificador único, utilizado pelo sistema para identificar a tarefa.
- **Endereço da tarefa:** ponto de entrada da tarefa.
- **Tipo de cancelamento:** como especificado pela API da POSIX, uma *thread* pode ter o cancelamento ativo ou desativo. Quando o cancelamento se encontra ativo este pode ser deferido - a *thread* apenas será cancelada quando

entra num ponto de cancelamento - ou assíncrono - o cancelamento acontece imediatamente.

- ***Detach state*:** As *threads* podem ser criadas *joinable* - a *thread* tem de esperar que todas as *threads* que criou como *joinable* terminem - ou *detached* - uma *thread* não tem de esperar que as *threads* criadas como *detached* terminem.
- **Estado da tarefa:** Sinaliza ao sistema se a tarefa se encontra adormecida ou não.
- **Retorno da tarefa:** valor de retorno a ser passado à tarefa mãe.
- ***Cleanup handlers*:** funções a serem executadas aquando da terminação da tarefa.

Quando uma tarefa é criada esta é adicionada à `PendingPCB` e é encapsulada dentro de outra função denominada de *wrapper*. Esta função é responsável por capturar o valor de retorno da tarefa, e lidar com a terminação da tarefa, sinalizando o escalonador que esta já não se encontra ativa, e que pode ser removida da `PendingPCB`.

3.2.5 Gestão de Recursos

Num sistema operativo com suporte a *multitasking* é necessário que este disponibilize um mecanismo capaz de assegurar a exclusão mútua. Contudo, ao contrário do que acontece num sistema *singlecore*, onde a exclusão mútua pode ser garantida simplesmente desabilitando e habilitando as interrupções, num sistema *multicore* tal já não é verdade, pois a desabilitação e habilitação das interrupções num dos *cores* do sistema não impede que outros acedam aos recursos que deviam estar protegidos.

Nesse sentido, o ARM *microkernel* SMP utiliza o mecanismo de exclusão mútua denominado de *spinlock*. Os *spinlock* tiram proveito dos mecanismos disponibilizados pelo processador para ler e modificar um local de memória de forma atómica, garantindo assim que o acesso a uma secção crítica apenas seja concedido a um dos *cores* do sistema de cada vez. Tal como visto na secção 3.1.6, a arquitetura ARM disponibiliza um par de instruções, que permitem a implementação de acessos atómicos à memória, sendo estas instruções utilizadas na implementação do

spinlock.

```
1  spin_lock_s
2     MOV     r2, #LOCK
3     LDREX   r1, [r0]
4     TEQ     r1, #UNLOCK
5     WFENE
6     STREXEQ r1, r2, [r0]
7     TEQEQ   r1, #0x0
8     BNE     spin_lock_s
9     BX      lr
10
11 spin_unlcok_s
12    MOV     r1, #UNLOCK
13    STR     r1, [r0]
14    MCR     p15, 0, r1, c7, c10, 4
15    SEV
16    BX      lr
```

Listagem 3.1: Implementação do *spinlock*

Na listagem 3.1 é apresentada a implementação do *spinlock* utilizada no ARM *microkernel* SMP. Na linha 2 o `ldrex` é utilizado para adquirir o valor do *lock*. Se o valor do *lock* for 1 significa que este já se encontra adquirido, e portanto, o *core* é posto em espera com o uso da instrução `wfe` (*wait-for-event*). Caso o valor seja 0, tenta adquirir o *lock* (linha 6), caso a instrução `strex` falhe, volta ao início para tentar, novamente, adquirir o *lock*. A função `spin_unlock_s` liberta o *lock* de forma incondicional alterando o valor do *lock* para 0, e posteriormente, utiliza a instrução `sev` (*send-event*) com o intuito de desbloquear todos os *cores* que se encontrem bloqueados na instrução `wfe`.

3.2.6 Sincronismo

Geralmente as aplicações desenvolvidas para sistemas operativos *multitask* encontram-se divididas em tarefas. Estas tarefas podem trabalhar cooperativamente, com o intuito de resolver problemas de complexidade elevada, necessitando assim, de comunicarem entre si para sincronizarem as suas atividades. Nesse sentido, os sistemas operativos disponibilizam mecanismos de sincronização e comunicação como

os *mutexs*, *conditional variables*, *semaphores* e *message queues*.

No caso do ARM *microkernel* SMP, são disponibilizados três mecanismos de sincronização:

- ***Mutexs***: garantem a serialização dos acessos a um recurso partilhado. Para que uma tarefa possa aceder a um recurso partilhado, primeiro tem de adquirir o *mutex* referente a esse recurso. Caso este já se encontre adquirido a tarefa irá esperar até que o *mutex* lhe seja atribuído. Nesta implementação, os *mutexs* funcionam como os *spinlocks* (3.2.5).

- ***Semaphores***: restringem o número máximo de acessos simultâneos a um recurso partilhado. Estes são implementados com recurso a um contador, e enquanto este for maior do que zero irá conceder o acesso às tarefas do recurso partilhado que protege. Uma tarefa pode requerer acesso ao recurso decrementado o contador, e sinalizar que já acabou de o utilizar incrementado o contador.

- ***Conditional variables***: permitem que uma tarefa aguarde até que uma condição seja verdadeira. Quando a tarefa se encontra em espera, esta é suspendida até que outra tarefa sinalize a *conditional variable*, tornando a condição verdadeira. Nesta implementação, este é o único mecanismo de sincronização onde as tarefas são efetivamente suspendidas.

3.3 Ambiente de Desenvolvimento

O propósito principal da presente dissertação é a exploração das potencialidades dos processadores *multicore* no desenvolvimento de sistemas operativos. Nesse sentido, uma das plataformas de desenvolvimento escolhidas, inicialmente, foi a Xilinx Zynq-7000, pelo fato de esta possuir um processador ARM Cortex-A9 *dualcore* e ferramentas de suporte capazes de simplificar tanto o desenvolvimento como a depuração. A outra plataforma de desenvolvimento escolhida foi a Versatile Express (disponibilizada virtualmente), pois esta pode ser instanciada com um processador ARM Cortex-A9 *quadcore*, indo assim de encontro a um dos principais requisitos do sistema a ser desenvolvido.

Intrinsecamente ligado às duas plataformas escolhidas encontram-se as suas ferramentas de desenvolvimento e depuração. No caso da Zynq-7000 foi utilizado o

ambiente de desenvolvimento da Xilinx (Xilinx ISE Design Suite). No caso da Versatile Express, foi também utilizado o Xilinx ISE Design Suite para o desenvolvimento de todo o código, mas com o fim de realizar a depuração do código foi utilizado o emulador da ARM denominado de Fast Models.

3.3.1 Xilinx Zynq-7000

A Zynq-7000 é baseada na arquitetura *Xilinx All Programmable SoC*. Esta integra no mesmo SoC um ARM Cortex-A9 *dual-core* (PS - *Processing System*) com lógica programável (PL - *Programmable Logic*) de 28 nm. O ARM Cortex-A9 é o elemento principal do PS, contudo este também inclui memória *on-chip*, interface para memórias externas e várias interfaces de conectividade para periféricos.

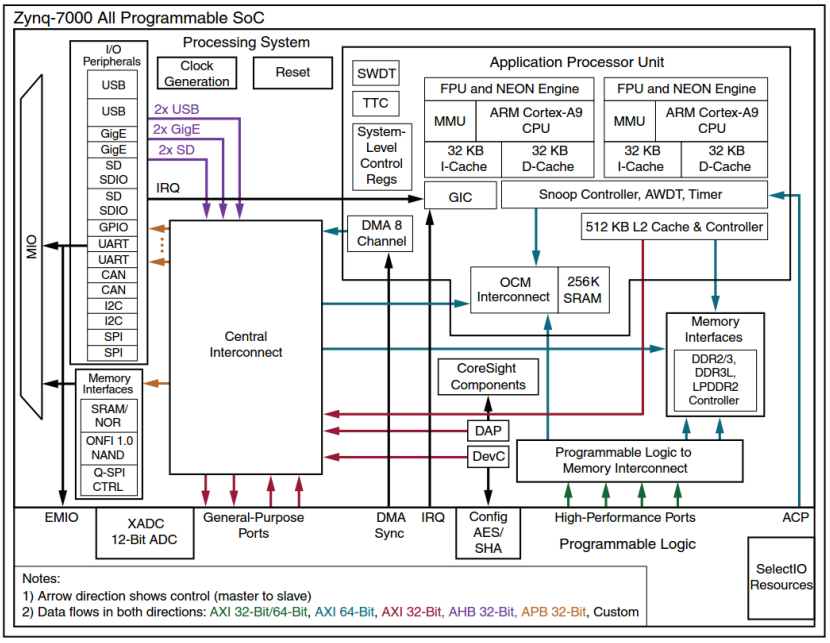

Figura 3.9: Zynq-7000 All Programmable SoC: diagrama de blocos [3]

A arquitetura da Zynq-7000 (figura 3.9) é capaz de oferecer o melhor de dois mundos, *hardware* e *software*, pois oferece a flexibilidade e escalabilidade das FPGAs, conjugada com a *performance* e facilidade de uso associado aos ASICs (*Application-Specific Integrated Circuits*). Na Zynq-7000 o PS é o componente principal do sistema, sendo o primeiro a ser inicializado durante o processo de *boot*, capaz de executar vários sistemas operativos (e.g. Linux) independentemente do uso da PL. A configuração da PL é controlada por *software* que corre no processador da PS.

3.3.2 Versatile Express (VE)

A Versatile Express é uma plataforma de desenvolvimento produzida pela ARM. Na presente dissertação foi utilizada uma versão virtual desta plataforma, nomeadamente a VE FVP [5] (*Versatile Express Fixed Virtual Platform*). Esta é um modelo em *software* do sistema implementado em *hardware* constituído por: (i) uma implementação virtual da *motherboard*; (ii) uma *daughterboard* por cada processador ARM; (iii) interligações associadas.

A VE FVP oferece uma execução de *software* funcionalmente correta. Contudo, é sacrificado a precisão temporal de execução em prol da velocidade de execução. Além disso, esta também possui outras divergências da versão de *hardware* como: (i) precisão temporal aproximada; (ii) barramentos simplificados; (iii) não possui a implementação das *caches* do processador e respetivos *buffers* de escrita. A VE FVP também é disponibilizada com vários processadores da família ARM, tendo sido escolhido na presente dissertação uma implementação com um Cortex-A9 *quadcore*.

3.3.3 Xilinx ISE Design Suite

Desenvolvido pela Xilinx, o ISE Design Suite é um ambiente de desenvolvimento, distribuído em três edições:

- ***Logic Edition:*** permite ir desde o *design* de um sistema até à implementação, validação e programação do dispositivo.
- ***Embedded Edition:*** inclui todas as ferramentas e capacidades da *Logic Edition*, acrescido das capacidades do EDK (*Embedded Development Kit*).
- ***DSP Edition:*** também inclui todas as funcionalidades da *Logic Edition*, mas para além disso, também inclui as funcionalidades do *System Generator* para DSP e o *AccelDSP Synthesis Tool*.

As ferramentas presentes no EDK foram as utilizadas para o desenvolvimento do sistema, tanto para a Zynq-7000 como para a VE.

Na figura 3.10 é ilustrado, de forma simplificada, as ferramentas do *Embedded Edition* utilizadas nesta dissertação, bem como estas se relacionam. Assim sendo, o PlanAhead é a ferramenta principal no desenvolvimento de um sistema embebido.

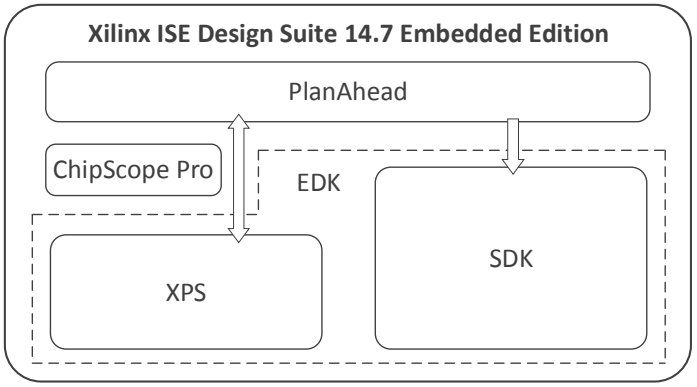

Figura 3.10: Diagrama de blocos - Xilinx ISE 14.7

Esta é responsável por interligar o hardware gerado pelo XPS (*Xilinx Platform Studio*) ao software desenvolvido no SDK (*Software Development Kit*). Ambas as ferramentas mencionadas encontram-se integradas no EDK.

***Embedded Development Kit* (EDK)**

O EDK é um ambiente de desenvolvimento integrado para desenvolver sistemas embebidos. Este providencia ferramentas fundamentais, tecnologias e um *design flow* familiar, para atingir resultados ótimos no desenvolvimento de sistemas embebidos. Nas ferramentas disponibilizadas estão incluídos o XPS e o SDK, bem como blocos IP e respetiva documentação.

- ***Xilinx Platform Studio* (XPS):** disponibiliza um ambiente integrado para o desenvolvimento, conexão e configuração de processadores embebidos, desde uma máquina de estados a um microcontrolador RISC de 32bits.
- ***Software Development Kit* (SDK):** providencia um ambiente de desenvolvimento de aplicações em C/C++.

3.3.4 ARM Fast Models

O Fast Models é um emulador desenvolvido pela ARM com o intuito de permitir o desenvolvimento de *software* em plataformas virtuais, mesmo antes da plataforma física correspondente se encontrar disponível. Esta oferece um portefólio dos mais recentes IP *cores* da ARM totalmente validados. Para além disso, ainda é possível

criar novas plataformas virtuais através do *System Canvas*, e simuladores para validar todo o sistema com o *System Generator*.

A nível de funcionalidades o Fast Models é capaz de: (i) executar até 250 milhões de instruções por segundo; (ii) adaptar o desempenho e precisão conforme a aplicação; (iii) arrancar qualquer sistema operativo em segundos (e.g. Linux, Android, Windows Embedded CE e Symbian); (iv) oferecer modelos de ISAs funcionalmente precisos e validados; (v) modelar tecnologias avançadas da ARM como *caches*, MMU, virtualização, TrustZone e VFP (*Vector Floating Point*); e (vi) incluir componentes virtualizados como Ethernet e LCD.

O ARM Fast Models também oferece interfaces de depuração e *trace* bastante completas e simples de usar. Estas interfaces, oferecem funcionalidades de depuração como a realização de *step*, *step in*, *step out*, *step over* e *breakpoints*. Além disso, também permitem a visualização do *disassembly* do código, registos do processador e da memória, e possui suporte para a realização de depuração *multicore*.

Capítulo 4

Desenvolvimento do sistema

Após a especificação do sistema no capítulo anterior, cabe ao presente capítulo descrever o desenvolvimento e implementação dos componentes constituintes do sistema final. O capítulo anterior permitiu a familiarização com a arquitetura do processador utilizado pelas plataformas de desenvolvimento alvo, o sistema operativo ARM *microkernel* SMP e as *toolchains* utilizadas. Este capítulo encontra-se dividido em três grandes subsecções, relacionadas com as três arquiteturas de multi-processamento desenvolvidas.

Primeiramente, é explicado o *porting* do ARM *microkernel* SMP para as plataformas de desenvolvimento VE e Zynq-7000. Nesta mesma fase é também analisado o sistema de gestão de memória dinâmica implementado. E por fim são apresentadas algumas das correções e alterações que foi necessário realizar às APIs do ARM *microkernel* SMP.

Na segunda fase, é explicado o processo de *refactoring* do ARM *microkernel* SMP, para uma arquitetura AMP. Aqui, é inicialmente analisado o desenvolvimento de uma versão *singlecore* deste e posteriormente é descrito o processo de integração, de várias cópias da versão *singlecore*, num sistema AMP.

Por fim, no final do capítulo são analisadas as várias etapas envolvidas no desenvolvimento do sistema H^2MP. Inicialmente, são analisadas as APIs de suporte implementadas, que visam possibilitar e facilitar a coexistência de vários sistemas operativos no mesmo sistema. De seguida é explicada a combinação das versões SMP e AMP do *microkernel* num sistema HMP. Por último é analisada a integração do ARM *microkernel* HMP, desenvolvido nesta fase, com o FreeRTOS num sistema de multi-processamento heterogéneo denominado de H^2MP.

4.1 *Porting* do ARM *microkenel* SMP

O *porting* do ARM *microkenel* SMP para as plataformas de desenvolvimento zynq-7000 e VE foi uma etapa essencial no início do desenvolvimento do sistema. Apesar destas plataformas possuírem processadores baseados na arquitetura do ARM Cortex-A9, existem caraterísticas diferentes entre estas, como por exemplo o sistema de memória e os periféricos que estas possuem. Sendo assim, foi necessário alterar os *device drivers* dos periféricos, a configuração do sistema de memória e além disso também foi necessário refazer a *linker script*. Apesar do código original ter sido desenvolvido na *toolchain* da ARM, e nesta dissertação ter sido utilizada a *toolchain* da GNU, não foi necessário a realização desta migração, pois já tinha sido feita anteriormente por um investigador do ESRG.

4.1.1 *Linker script*

O *linker* é a ferramenta responsável por gerar o ficheiro executável final (output file) a partir dos ficheiros objeto gerados pelo compilador (input files). O *linker* é sempre controlado por uma *linker scprit*, e esta por sua vez tem como propósito descrever como as sessões dos ficheiros de entrada são mapeados no ficheiro final e o *layout* deste na memória.

Nesse sentido, foi necessário refazer a *linker script* do ARM *microkenel* SMP, para ser possível configurar o *layout* deste no sistema de memória das plataformas utilizadas. Nesta secção apenas serão analisadas os segmentos desta comuns entre as plataformas alvo, sendo posteriormente descrito, nas secções seguintes, as partes intrinsecamente ligadas ao sistema de memória de cada plataforma de desenvolvimento utilizada.

```
1 /* NUMERO DE CPUS */
2 N_CPUS = 4;
3
4 /* TAMANHO DAS STACKS */
5 IRQ_STACK_SIZE = 16k * N_CPUS;
6 SVC_STACK_SIZE = 16k * N_CPUS;
7 SYS_STACK_SIZE = 16k * N_CPUS;
8 ABORT_STACK_SIZE = 16k * N_CPUS;
```

```
9  UNDEF_STACK_SIZE = 16k * N_CPUS;
10
11 USER_STACK_TOP = MEMORY_TOP - (SYS_STACK_SIZE + IRQ_STACK_SIZE +
       SVC_STACK_SIZE + ABORT_STACK_SIZE + UNDEF_STACK_SIZE);
12
13 /* TAMANHO DA HEAP */
14 HEAP_SIZE = 1M;
```

Listagem 4.1: Definição dos símbolos de configuração

Na listagem 4.1 é apresentada a porção da *linker script* onde são definidos os símbolos responsáveis por configurar a quantidade de memória atribuída às *stacks* e à *heap*. Como analisado na secção 3.2, o ARM *microkenel* SMP utiliza os modos SVC, IRQ e SYS logo, é necessário reservar memória para as *stacks* de cada um destes modos. Contudo, como este utiliza mais do que uma unidade de processamento, e cada uma destas tem o seu próprio *register file*, torna-se necessário alocar espaço para as *stacks* utilizadas de cada unidade de processamento, que constitui o sistema SMP. Nesse sentido, é especificado no símbolo `N_CPUS` o número de unidades de processamento utilizadas para que se possa configurar a quantidade de memória alocada para as *stacks* de acordo com esse número. Além disso, nesta mesma secção é também especificado a quantidade de memória que será utilizada para implementar a memória dinâmica (linha 12). Por último, toda a memória que se encontrar livre é utilizada para instanciar as *stacks* das *threads* do utilizador (linha 9).

```
1  /* PONTO DE ENTRADA */
2  ENTRY(_start)
3
4  /* SECCOES */
5  SECTIONS
6  {
7    /* Start up code and text */
8    .__startup_section : ALIGN(4) {
9      _STARTUP_START = .;
10       src/arch/boot.o (.text);
11       src/*(.text);
12     _STARTUP_END = .;
13   } > DDR
```

```
14
15    /* MMU L1 page table */
16    .page_table_section : ALIGN(16k) {
17      _PAGE_TABLE_START = .;
18        src/arch/mmu.o (.data);
19      _PAGE_TABLE_END = .;
20    } > DDR
21
22    /* {...} */
23  }
```

Listagem 4.2: Secção do código e da MMU *translation table*

Na *linker script* é também definido onde o código deve começar a ser executado. Para tal, foi utilizado o comando `ENTRY` onde é especificado que o ponto de entrada do código é no *_start*, que coincide com o início do código de inicialização contido no ficheiro `boot.S`. Além do ponto de entrada, na *linker script* desenvolvida também é especificado o local onde as diferentes secções de memória, constituintes do código, são colocadas na memória. Como se pode observar na listagem 4.2 linha 10, a primeira secção especificada foi a secção que contém todo o código do sistema operativo (`.text`), sendo também especificado que o código constituinte do ficheiro `boot.S` será o primeiro a aparecer, devido a ser este o primeiro código a ser executado. A seguir a esta secção foi colocada a secção que possui a *translation table* da MMU, sendo especificado que esta comece num endereço com um alinhamento de 16 *kbytes* - requisito essencial da MMU.

Para além das secções apresentadas na listagem 4.2, também são especificadas as secções relacionadas com as variáveis (`.rodata`, `.data` e `.bss`), a memoria dinâmica e as *stacks*. As secções destinadas às *stacks* foram todas alocadas em endereços com um alinhamento de 8 *bytes*, tal como é especificado no EABI (*Embedded Application Binary Interface*) da ARM.

4.1.2 *Porting* para a Zynq-7000

O *porting* realizado para a Zynq-7000 consistiu na alteração da configuração da memória, na especificação de uma *translation table* para a MMU, e também na implementação de um *device driver* para a UART.

Sistema de memória

O *layout* do sistema de memória da Zynq-7000 encontra-se apresentado na figura 4.1. Como se pode observar nesta figura, a memória RAM (*random access memory*), disponível para ser utilizada pelo sistema operativo, tem um tamanho de 1 *gigabyte* e começa no endereço zero. Os restantes 3 *gigabytes* de memória são destinados aos periféricos constituintes do sistema.

Start Address	Size (MB)	Description
0x0000_0000	1,024	DDR DRAM and on-chip memory (OCM)
0x4000_0000	1,024	PL AXI slave port #0
0x8000_0000	1,024	PL AXI slave port #1
0xE000_0000	256	IOP devices
0xF000_0000	128	Reserved
0xF800_0000	32	Programmable registers access via AMBA APB bus
0xFA00_0000	32	Reserved
0xFC00_0000	64 MB - 256 KB	Quad-SPI linear address base address (except top 256 KB which is in OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

Figura 4.1: Mapeamento da memória da Zynq-7000 [4]

Nesse sentido, na *linker script*, para a Zynq-7000 foi especificado utilizando o comando `MEMORY`, o espaço de memória disponível, de acordo com a memória RAM disponibilizada pela Zynq-7000. Essa especificação é apresentada na listagem 4.3, onde é possível observar na linha 7 a configuração do espaço de memória disponível.

```
1 /* MEMORIA */
2 MEMORY_TOP = 0x3FF00000;
3
4 MEMORY
5 {
6   OCM_RAM0 (rwx) : ORIGIN = 0x00000000, LENGTH = 0x00030000
7   DDR (rwx) : ORIGIN = 0x00100000, LENGTH = 0x40000000
8   OCM_RAM1 (rwx) : ORIGIN = 0xFFFF0000, LENGTH = 0x0000FE00
9 }
```

Listagem 4.3: Especificação dos limites da memória

Além disso, na *linker script* é também definido o símbolo `MEMORY_TOP`, que possui o endereço do topo da memoria menos 1 *megabyte*. Na implementação original do *microkernel* é definido que o último *megabyte* da memória é utilizado na sincronização da inicialização do sistema SMP, não podendo nesse sentido, ser utilizado por outras secções de memória.

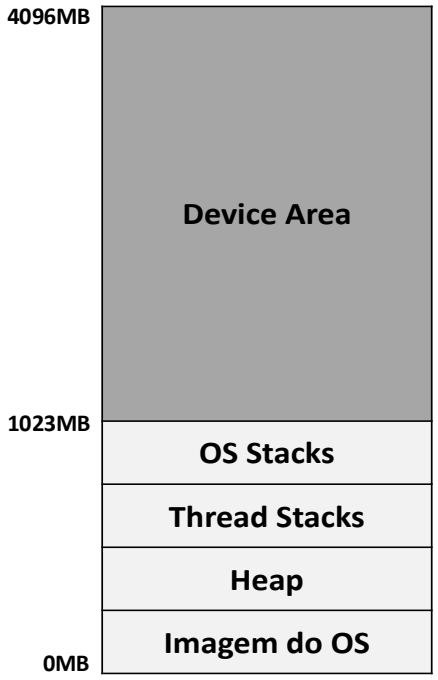

Figura 4.2: Configuração da memória da Zynq-7000

Por fim, a *translation table* da MMU foi alterada de forma a ir de encontro às especificações do sistema de memória, e aos requisitos do sistema operativo. A configuração da memória para a Zynq-7000 está apresentada na figura 4.2. Esta encontra-se dividida em dois grandes blocos:

1. **Região de memória do sistema operativo**: primeiros 1023 *megabytes*. Configurada como *Normal Shared memory* onde se encontram todas as secções de memória constituintes do sistema operativo, especificadas anteriormente na *linker script*.

2. **Região de memória dos periféricos**: restante memória. Configurada como *device memory*.

4.1.3 *Porting* para a VE

O *porting* realizado para a VE incidiu nas mesmas alterações realizadas para a plataforma Zynq-7000. Nesse sentido, as secções de códigos referentes à configuração da memória (e.g. *translation table* da MMU) e à *device driver* da UART, neste caso a PL011, tiveram de ser refeitos.

Sistema de memória

Peripheral	Modeled	Address range	Size
NOR FLASH0 (CS0)	Yes	0x00_00000000–0x00_03FFFFFF	64MB
Reserved	-	0x00_04000000–0x00_07FFFFFF	64MB
NOR FLASH0 alias (CS0)	Yes	0x00_08000000–0x00_0BFFFFFF	64MB
NOR FLASH1 (CS4)	Yes	0x00_0C000000–0x00_0FFFFFFF	64MB
Unused (CS5)	-	0x00_10000000–0x00_13FFFFFF	-
PSRAM (CS1) - unused	No	0x00_14000000–0x00_17FFFFFF	-
Peripherals (CS2)	Yes	0x00_18000000–0x00_1BFFFFFF	64MB
Peripherals (CS3)	Yes	0x00_1C000000–0x00_1FFFFFFF	64MB
CoreSight and peripherals	No	0x00_20000000–0x00_2CFFFFFF[a]	-
Graphics space	No	0x00_2D000000–0x00_2D00FFFF	-
System SRAM	Yes	0x00_2E000000–0x00_2EFFFFFF	64KB
Ext AXI	No	0x00_2F000000–0x00_7FFFFFFF	-
4GB DRAM (in 32-bit address space)[b]	Yes	0x00_80000000–0x00_FFFFFFFF	2GB

Figura 4.3: Mapeamento da memória da VE [5]

A organização do sistema de memória da VE encontra-se exposto na figura 4.3. Nesta figura é possível observar que a memória RAM encontra-se mapeada nos últimos 2 *gigabytes* de memória (endereço base: 0x80000000, endereço topo: 0xFFFFFFF). Os restantes endereços de memória estão relacionados com os periféricos de *hardware* e memórias não voláteis.

```
1 /* MEMORIA */
2 MEMORY_TOP = 0xFFF00000;
3
```

```
4 MEMORY
5 {
6   /* RAM */
7   DDR (rwx) : ORIGIN = 0x80000000, LENGTH = 0x80000000
8 }
```

Listagem 4.4: Especificação dos limites da memória

A secção da *linker script* onde os limites da memória são definidos também teve de ser alterada, com os parâmetros da memória RAM da VE. Sendo assim, como se pode observar na listagem 4.4, o *linker*, para a VE, pode utilizar todo o espaço de memória desde o endereço 0x80000000 até ao endereço 0xFFFFFFF. Tal como acontecia na *linker script* da Zynq-7000, e pela mesma razão, o símbolo `MEMORY_TOP` possui o valor do topo da memória menos 1 *megabyte*.

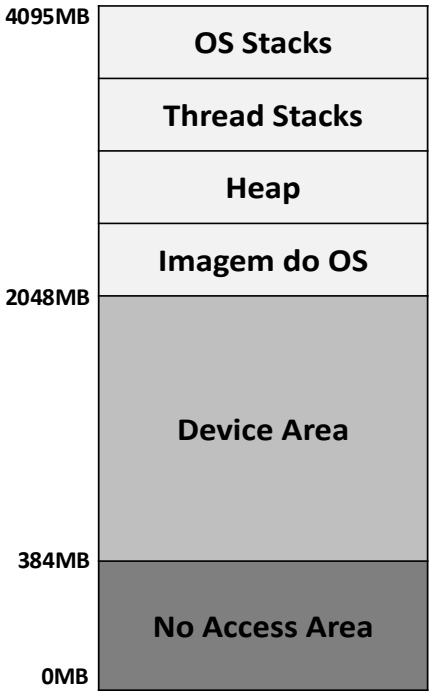

Figura 4.4: Configuração da memória da VE

Por fim, foi refeita a *translation table* da MMU, com base no *layout* da memória apresentado na figura 4.3. Assim, a configuração da memória da VE, utilizada pelo sistema operativo, apresenta a disposição apresentada na figura 4.4. Conforme se pode constatar na figura, a memória encontra-se dividida em três secções:

1. **Memórias não voláteis**: desde o endereço 0x0 até ao endereço 0x17FFFFFF.

Configurada como memória sem acesso pois esta não é utilizado pelo sistema operativo.

2. **Memória dos periféricos**: desde o endereço 0x18000000 até ao endereço 0x7FFFFFFF. Configurada como *device memory*.

3. **Memória do sistema operativo**: últimos 2047 *megabytes*. Configurada como *Normal Shared memory* onde se encontram todas as secções de memória constituintes do sistema operativo.

4.1.4 Implementação da memória dinâmica

O ARM *microkernel* SMP possui funcionalidades que necessitam de alocar memória de forma dinâmica. Nesse sentido, foi necessário a implementação de uma *heap*, de forma a dar suporte às necessidades de alocação dinâmica de memória.

A *heap* implementada consiste num bloco de memória constituído por blocos de menores dimensão, de memória alocada e memória livre. Cada bloco de memória possui um cabeçalho, que contém informação sobre o tamanho do bloco, e um apontador para o bloco livre ou alocado seguinte.

Inicialmente, a *heap* é constituída por um único bloco de memória livre, do tamanho da *heap*. O primeiro bloco livre é sempre apontado pelo apontador `FREE_LIST`. Quando é requerida uma alocação, é procurado na lista de blocos livres, um bloco capaz de satisfazer as necessidades do pedido realizado. Caso o bloco encontrado seja maior do que é pretendido, o bloco é partido em dois. Assim, a *heap*, que inicialmente era apenas constituída por um único bloco, pode tornar-se numa lista ligada de vários blocos de menores dimensões livres, intercalados por vários blocos de memória alocada.

Além disso, a *heap* implementada utiliza uma política de alocação *first fit*. Assim, esta sempre que recebe um pedido de alocação de memória, irá devolver o primeiro bloco livre com um tamanho suficiente para satisfazer o pedido, dividindo o bloco caso este seja maior do que é pretendido. Os blocos livres, por sua vez, são organizados pelos seus endereços, facilitando assim, a junção dos blocos livres contíguos.

A figura 4.5 mostra o estado inicial *heap*, e o estado desta após terem sido realizadas alocações e libertações de memória sobre esta. O apontador `FREE_LIST`, após terem

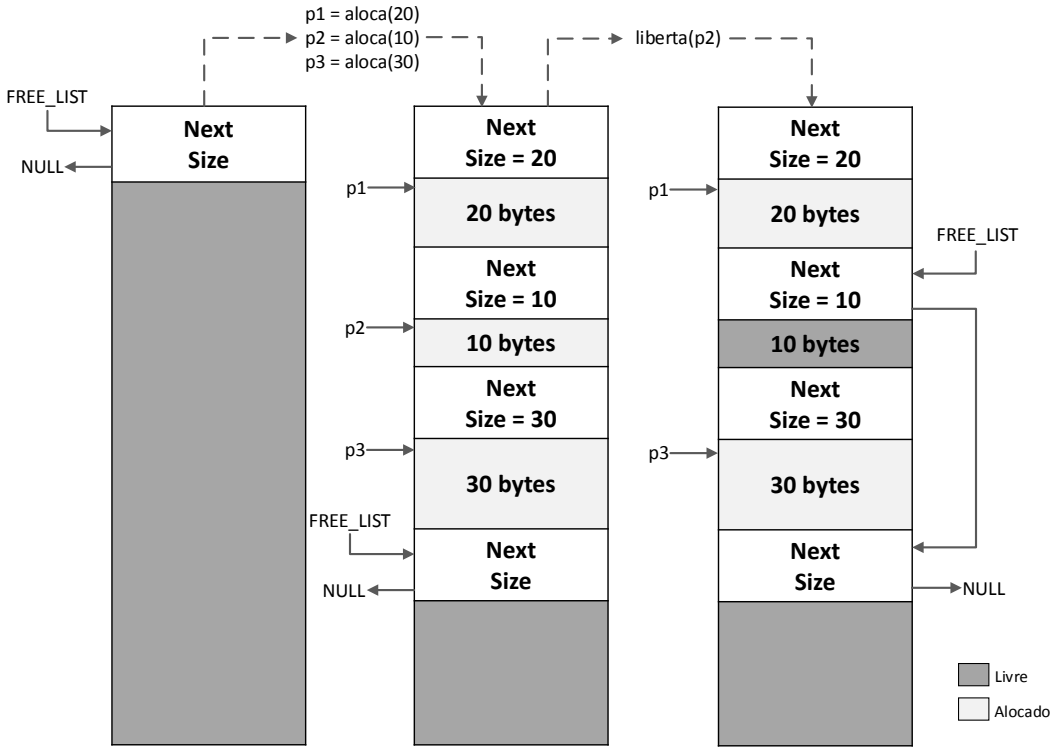

Figura 4.5: Funcionamento da *heap*

sido realizadas as alocações de 20 *bytes*, 10 *bytes* e 30 *bytes*, passa a apontar para o primeiro bloco de memória livre disponível. Seguidamente, após a libertação do bloco de memória alocado ao apontador `p2`, esse bloco fica livre, e passa a ser apontado pelo apontador `FREE_LIST` pois, este aparece primeiro que o segundo bloco livre, que passa por sua vez a ser apontado pelo pelo apontador `next` do primeiro bloco livre. Caso fosse realizada, por exemplo, uma nova alocação de 4 *bytes*, o primeiro bloco livre seria dividido num bloco de 4 bytes e noutro com a restante memória. Se o bloco, que possui a restante memória, for demasiado pequeno para satisfazer algum pedido, o bloco inicial não é dividido sendo assim, retornado um bloco de memória ligeiramente superior ao requerido.

4.1.5 Teste e correção das APIs

Após a realização dos *portings* e implementação da memória dinâmica, sendo esta requerida por algumas funcionalidades constituintes do ARM *microkernel* SMP, realizaram-se testes às APIs do *microkernel* disponibilizado pela ARM, de forma a averiguar o correto funcionamento destas. Contudo, durante estes testes ficou claro

que algumas das APIs necessitavam de ser alteradas pois, não apresentavam um comportamento correto. Sendo assim, as correções efetuadas foram as seguintes:

1. Na configuração da MMU foram alteradas as permissões de acesso de todos os domínios de *manager* para *client*, para que as restrições dos acessos ao sistema de memória sejam cumpridas.

2. Na função `__threads_init` foi corrigido a inicialização das `pcbs` das *threads*.

3. Nas funções `pthread_setcancelstate` e `pthread_setcanceltype` foi corrigido a verificação dos atributos passados como parâmetros.

4. Na função `pthread_create` foi corrigido a inicialização do semáforo responsável por sinalizar a finalização da *thread*, quando esta é criada como *joinable*. O semáforo não era inicializado para todos os casos necessários.

Além disso, também foi verificado que uma das *dummy threads* encontrava-se a ser executada, quando existiam *threads* do utilizador ativas. Isto devia-se ao facto de ser umas das *dummy threads* que executava a `main` do programa. Por esse motivo, essa *dummy threads* na sua criação não era marcada como tal, e consequentemente era selecionada para ser executada inadvertidamente. Como resolução foi adicionada a função `threads_creation_completed`, que deve ser invocada no fim da função `main`, que irá marcar a *thread* como sendo *dummy* e invocar o escalonador, para selecionar uma nova *thread* para ser executada.

4.2 *Refactoring* do ARM *microkernel* SMP para AMP

Ao contrário do que acontece na configuração SMP, na AMP existe um sistema operativo a correr em cada unidade de processamento. As vantagens desta abordagem assentam no facto de se poder conservar as caraterísticas originais do sistema operativo, possibilitando assim que as aplicações, previamente desenvolvidas para estes, possam ser executadas mantendo o seu comportamento original, sem a necessidade de as alterar. Além disso, os sistemas operativos podem conservar os seus mecanismos de exclusão mútua não sofrendo assim, do *overhead* acrescido pela sincronização de aplicações, a correr em diferentes unidades de processamento.

Contudo, esta abordagem apresenta uma grande penalização no uso de memória,

pois ao invés de existir apenas uma imagem de um sistema operativo, passa a existir uma por cada unidade de processamento. Outro aspeto importante a ter em conta é o facto de sem a adição de mecanismos transversais ao sistema global, não é possível comunicar nem sincronizar as aplicações que correm em diferentes sistemas operativos, e mesmo no caso da existência destes mecanismos, estes serão sempre mais custosos do que os existentes num sistema SMP.

Assim sendo, o *refactoring* do *microkernel* pode ser dividido em duas fazes distintas. A primeira fase, assenta essencialmente na alteração do *microkernel* para uma versão *singlecore*, onde serão revistos mecanismos como o processo de inicialização, sincronismo e o escalonador. A segunda fase, relaciona-se exclusivamente com a construção do sistema sistema AMP *quad-core* homogéneo, tendo em consideração aspetos como a necessidade, na fase de arranque do sistema, de um dos sistemas operativos ser responsável por lançar os restantes sistemas operativos, a coexistência de recursos partilhados e a distribuição da memória.

4.2.1 ARM *microkernel singlecore*

A versão original do *microkernel* foi implementada com o intuito de ser executada em várias unidades de processamento em simultâneo (abordagem SMP). Nesse sentido, este possui no seu código várias secções que se destinam a dar suporte a esse tipo de configuração. No entanto, a versão *singlecore* não necessita desse suporte e mesmo algumas implementações, como a dos mecanismos de sincronização, necessitam de ser refeitas, pois o que fazia sentido numa implementação SMP pode não fazer numa implementação *singlecore*. Seguidamente, serão então analisadas as alterações efetuadas, de forma a passar de uma implementação *multicore* para uma *singlecore*. Os mecanismos transversais de comunicação e sincronização serão abordados posteriormente na secção 4.3;

Boot

O código de arranque original foi implementado de forma a dar suporte e a possibilitar uma execução no modo SMP. Por isso, várias secções do código de *boot* já não são necessárias, para inicializar uma versão *singlecore* do *microkernel*, onde apenas uma única unidade de processamento é responsável por gerir todo o sistema. Assim, como se pode ver na figura 4.6, o código do *boot* já não tem nenhuma preocupação como a sincronização do processo de arranque, e além disso é responsável

por inicializar todo o processador (pilhas, MMU, sistema de memória) e todas as bibliotecas de suporte. No que diz respeito aos recursos de *hardware* que se destinam a dar suporte ao processamento paralelo, como a SCU, já não são utilizados, pois as funcionalidades destes já não têm utilidade nesta implementação.

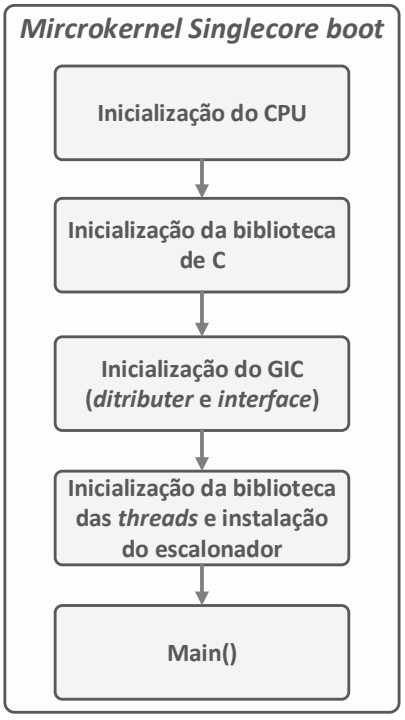

Figura 4.6: Processo de arranque do ARM *microkernel singlecore*

Escalonador

No que diz respeito ao escalonador, este em si não sofreu grandes alterações. Apenas foram retirados os mecanismos de exclusão mútua utilizados, para garantir o acesso exclusivo a este, pois como o *microkernel* só é executado por uma única unidade de processamento, este não precisa ser protegido contra acessos concorrentes. No entanto, a geração do *tick* que desponta a execução deste teve de sofrer alterações mais significativas. Isto porque na versão original como apenas era utilizado um temporizador, existia a necessidade de adicionar um mecanismo que distribuísse o *tick* por todas as unidades de processamento. No entanto, na versão *singlecore* tal já não é necessário, pois a interrupção gerada pelo temporizador pode despontar diretamente a execução do escalonador. Este mecanismo pode ser observado na figura 4.7.

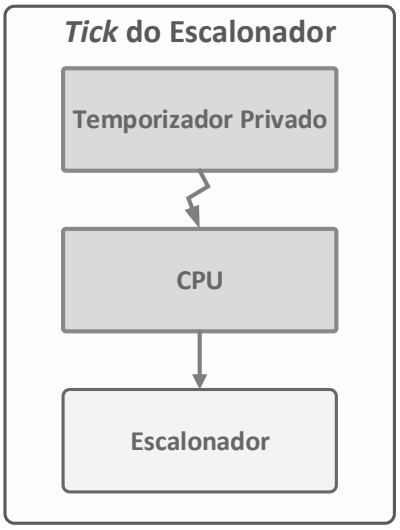

Figura 4.7: Geração do *tick* do escalonador

Sincronização

Os métodos de sincronização, em geral, requereram alterações mais significativas. Com o intuito de facilitar a compreensão das alterações efetuadas e as razões destas, seguidamente serão analisadas as novas implementações juntamente com as originais, de forma a relacionar as alterações efetuadas com o tipo de ambiente de execução a que se destinam (*singlecore* ou *multicore*).

Os métodos de sincronização que requereram uma reestruturação mais aprofundada foram:

1. **Primitiva básica de exclusão mútua**

   - **Implementação original:** implementada com recurso a *spinlocks*, que são capazes de garantir acessos exclusivos a secções de código crítico, num ambiente onde existe verdadeira concorrência.

   - **Implementação *singlecore*:** já não existe concorrência verdadeira, logo a proteção de secções críticas de código pode ser atingida garantindo que essas secções sejam executadas ininterruptamente. Assim sendo, nesta implementação é utilizada a desabilitação e habilitação das interrupções como forma de proteção das secções criticas.

2. ***Mutexs***

- **Implementação original**: funcionam como os *spinlocks*. Quando o *mutex* já se encontra adquirido as tarefas ficam bloqueadas num *spinlock*, até conseguirem adquirir o *mutex*. Nesta implementação, as tarefas não são suspendidas ficando ativas até ao fim do seu *time-slice* e podem ser resumidas mesmo sem terem adquirido o *mutex*, continuando assim bloqueadas no *spinlock*.

- **Implementação *singlecore***: ao contrário do que acontece na implementação *multicore*, só existe uma tarefa a ser executada em cada momento, não havendo portanto a possibilidade do *mutex* ser libertado enquanto esta se encontra a ser executada. Logo, foi implementado um mecanismo que permite a suspensão de tarefas bloqueadas nos *mutexs*, colocando estas no estado *sleep*. Posteriormente é utilizada uma política FIFO, para atribuir o *mutex* às tarefas bloqueadas.

3. ***Semaphores***

   - **Implementação original**: funcionam de forma semelhante aos *spinlocks*, com a diferença de em vez de testar se um *lock* se encontra bloqueado ou livre, utiliza um contador que enquanto este não atinge um valor negativo, concede acesso às tarefas. Tal como acontece nos *mutexs*, as tarefas bloqueadas são mantidas ativas, presas no *spinlock* do *semaphore*.

   - **Implementação *singlecore***: pelo mesmo motivo que levou à alteração da implementação dos *mutexs*, os *semaphores* também foram alterados para permitir a suspensão das tarefas. As tarefas bloqueadas são colocadas no estado *sleep* e posteriormente resumidas assim que um novo *post* seja executado, seguindo uma política FIFO.

Para além destas alterações, várias outras funções foram alteradas principalmente na forma como utilizavam os mecanismos de sincronização. Na implementação original, todas as secções criticas de código eram protegidas utilizando *mutexs*. Contudo, estas foram alteradas para usarem um mecanismo mais leve, a desabilitação e habilitação das interrupções. Seguidamente, são apresentadas e descritas todas as funções presentes na versão original que foram removidas na versão *singlecore*.

- `holding_pen`: utilizada na sincronização do processo de inicialização das várias unidades de processamento.

- `__cpu_idle`: função responsável por finalizar as inicializações locais das unidade de processamento secundárias. É também aqui que os outros processadores aguardam até receberem a IPI referente à função de escalonamento.
- `scheduler_timer`: funciona em conjunto com o temporizador responsável por gerar o *tick* do sistema. Esta função é responsável por enviar a IPI, referente ao escalonador, para todas as unidades de processamento seguindo uma política *round-robin*.
- `acquire_spinlock`: permite adquirir um *spinlock*.
- `acquire_trylock`: tenta adquirir um *spinlock*.
- `release_spinlock`: liberta um *spinlock*.

4.2.2 ARM *microkernel* AMP

Num sistema AMP existe mais do que um sistema operativo a correr concorrentemente no mesmo processador. Logo, estes têm de ser implementados de forma a possibilitar essa coexistência, e para além disso um dos sistemas operativos terá de ser responsável por arrancar os restantes sistemas operativos, e inicializar todos os componentes de *hardware* partilhados entre estes. Nesse sentido, a versão *singlecore* do ARM *microkernel* SMP teve de ser alterada de forma a ir de encontro aos requisitos mencionados. As alterações efetuadas incidiram essencialmente na configuração da memória e no arranque do sistema, sendo estas analisadas em maior detalhe seguidamente.

Configuração da memória

Como mencionado anteriormente, o *microkernel* recorre à MMU para configurar a memória disponível. Como na implementação original apenas existe um imagem do *microkernel*, a *translation table* é instanciada estaticamente e configurada de forma a utilizar toda a memória disponível. Contudo, na implementação AMP a memória tem de ser distribuída por mais do que uma imagem do *microkernel*, sendo nesse sentido necessário implementar um mecanismo que permita a configuração dinâmica da *translation table*.

Na listagem 4.5 é apresentada a função implementada para esse efeito. Esta, tendo em consideração os requisitos do *microkernel*, apenas permite a configuração da

translation table de nível 1 com um mapeamento direto. A função recebe como parâmetros um apontador para a *translation table* a ser configurada, o endereço base da secção de memória, o tamanho da secção e os atributos a serem atribuídos a essa secção de memória.

```
1  .global mmuRegionSet
2  .func mmuRegionSet
3  // void mmuRegionSet(uint32_t PT, uint32_t Base_addr, uint32_t
       size, uint32_t attr)
4  // r0 -> PT; r1 -> Base_addr; r2 -> size; r3 -> attr
5  mmuRegionSet:
6     add    r0, r0, r1, lsr #18
7     mov    r12, #0
8  mmu_loop:
9     cmp    r12, r2
10    beq    mmu_loop_end
11    push   {r2}
12    add    r2, r1, r12, lsl #20
13    orr    r2, r2, r3
14    stmia  r0!, {r2}
15    pop    {r2}
16    add    r12, r12, #1
17    b    mmu_loop
18 mmu_loop_end:
19    bx   lr
20 .endfunc
```

Listagem 4.5: Função utilizada para configurar a *translation table*

Ao contrário do que acontecia na versão original, durante o processo de inicialização do *microkernel*, este terá de configurar a sua *translation table*, utilizando a função `mmuRegionSet`, de acordo com os limites de memória atribuídos. Desta forma, cada imagem do *microkernel* irá configurar corretamente o seu espaço de memória, sem a necessidade de alterar o código fonte, garantido que este só tem acesso à memória que lhe é destinada. Na figura 4.8 é apresentada uma possível distribuição da memória num sistema AMP *quadcore* para a VE.

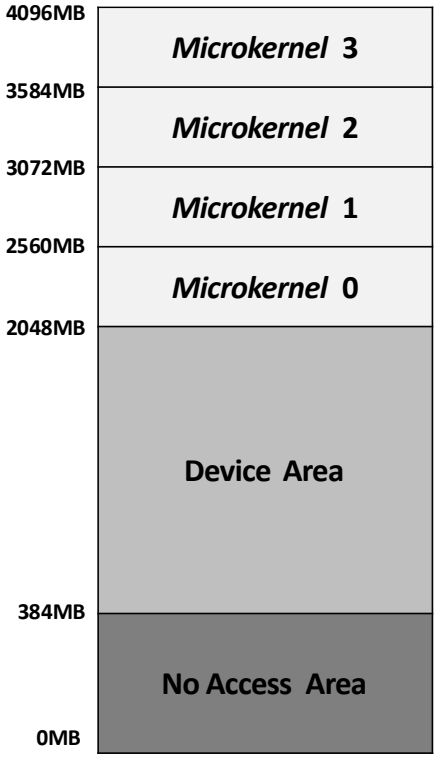

Figura 4.8: AMP *quadcore* configuração da memória

Boot do *microkernel* AMP

Quando o processador arranca, todos as unidades de processamento começam a executar instruções do mesmo endereço de memória. Contudo, num sistema AMP cada unidade de processamento executa imagens distintas do *microkernel*, logo torna-se necessário a existência de uma secção inicial de código responsável por encaminhar cada unidade de processamento para a sua respetiva imagem do *microkernel*. Além disso, os periféricos partilhados, como o distribuidor do GIC, apenas devem ser inicializados uma vez. Dessa forma, esse tipo de tarefas devem ser atribuídas ao Sistema Operativo primário.

Nesse sentido, foi adicionada ao código de inicialização do *microkernel* uma secção de código inicial que irá encaminhar todas as unidades de processamento para a sua respetiva imagem. Esta secção de código apenas se encontra presente no código do *microkernel* configurado como o OS primário. O OS primário será então responsável por gerir o arranque de todos os *microkernels* e de executar todas as inicializações dos periféricos partilhados, seguindo o esquema apresentado na figura 4.9.

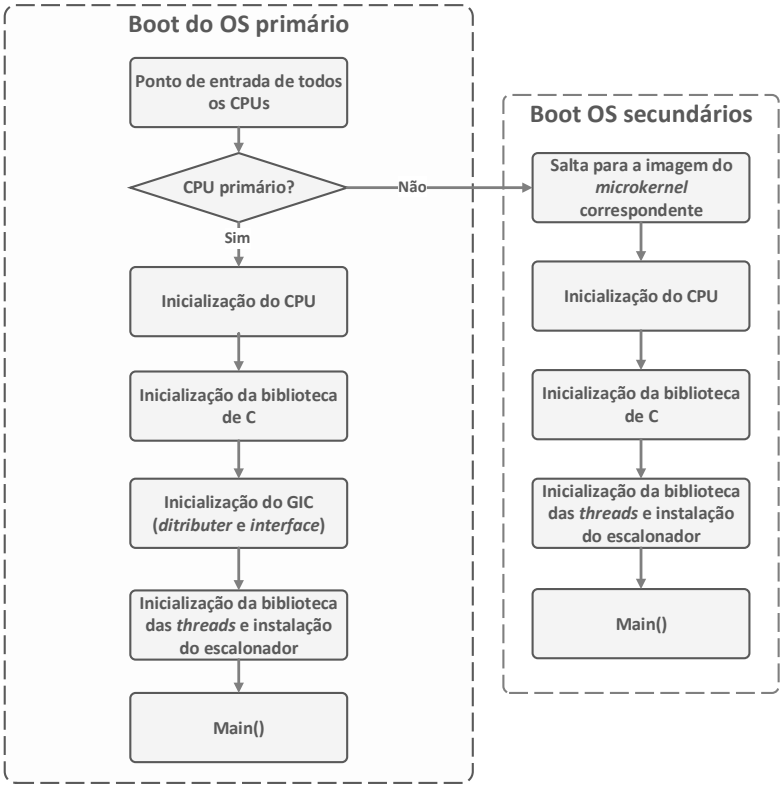

Figura 4.9: Inicialização do sistema AMP

Para além disso, é no código do OS primário que a inclusão das imagens dos restantes *microkernels* é realizada. Nesse sentido, no ficheiro `boot.S` foi utilizada a diretiva `.incbin` para adicionar os binários das restantes imagens ao projeto final (listagem 4.6). Para além disso, a *linker script* do OS primário também necessitou de ser alterada para possibilitar a especificação do local de memória em que cada imagem deve ser colocada (listagem 4.7).

```
1  .section OS_1, "a"
2  .global OS_1_start
3  .global OS_1_end
4  OS_1_start:
5     .incbin "OS_1.bin";
6  OS_1_end
7
8  .section OS_2, "a"
9  .global OS_2_start
10 .global OS_2_end
```

```
11 OS_2_start:
12    .incbin "OS_2.bin";
13 OS_2_end
14
15 .section OS_3, "a"
16 .global OS_3_start
17 .global OS_3_end
18 OS_3_start:
19    .incbin "OS_3.bin";
20 OS_3_end
```

Listagem 4.6: Inclusão das imagens dos *microkernels*

```
1  . = 0xA0000000;
2  _START_OS_1 = .;
3  OS_1 : {
4     *(OS_1);
5  }
6
7  . = 0xC0000000;
8  _START_OS_2 = .;
9  OS_2 : {
10    *(OS_2);
11 }
12
13 . = 0xE0000000;
14 _START_OS_3 = .;
15 OS_3 : {
16    *(OS_3);
17 }
```

Listagem 4.7: Especificação dos locais de memória onde cada imagem será colocada

4.3 *Hybrid and heterogeneous multiprocessing* H^2MP

A última fase de desenvolvimento consistiu no desenvolvimento de uma nova arquitetura de multi-processamento designada de HMP, na expansão desta para uma versão heterogénea (H^2MP), e finalmente no desenvolvimento dos mecanismos de suporte a estas arquiteturas. Nesse sentido, esta fase foi dividida em três etapas distintas. A primeira etapa relaciona-se com a implementação dos mecanismos de suporte, que possibilitam a coexistência e interação dos vários sistemas operativos presentes no sistema. Posteriormente, foi implementada a versão HMP do ARM *microkernel*, combinando as versões SMP e AMP, tendo como um dos principais objetivos permitir a configurabilidade deste nas diferentes configurações possíveis. Por fim, na última etapa foi implementado o H^2MP, combinando no mesmo sistema o ARM *microkernel* e o FreeRTOS, recorrendo aos mecanismos implementados na primeira etapa, para permitir a conjugação e coexistência de ambos no mesmo sistema.

4.3.1 Sincronização entre Sistemas Operativos

Como forma de possibilitar a coexistência de vários sistemas operativos no mesmo sistema é necessário ter em consideração determinados aspetos, como a partilha de recursos de *hardware* e de *software*. Nesse sentido, foi desenvolvido uma API de sincronização entre sistemas operativos que possibilita a sincronização do acesso a recursos partilhados, e também a sincronização entre tarefas a serem executadas em diferentes sistemas operativos.

A API implementada é baseada nos mecanismos de sincronização da POSIX (*mutexs*, *semaphores* e *conditional variables*). Além desses mecanismos, também são disponibilizadas funções que permitem a sincronização dos processos de inicialização, dos sistemas operativos, mesmo antes dos restantes mecanismos se encontrarem disponíveis. Assim, a API permite a sincronização dos processos de inicialização, dos acessos a recursos partilhados e a sincronização entre tarefas em diferentes sistemas operativos.

Outra caraterística relevante desta API reside no facto de ter sido desenvolvida de forma a poder ser facilmente integrada em qualquer sistema operativo. Nesse sentido, a API não especifica como os mecanismos de suspensão e resumo de tarefas,

para os mecanismos de sincronização global, devem ser geridos por cada sistema operativo, permitindo que estes implementem estes mecanismos, de acordo com os seus requisitos. Além disso, a API não tem conhecimento de quantas tarefas se encontram bloqueadas, por exemplo num *mutex* global, mas sim quais os sistemas operativos que possuem tarefas bloqueadas neste, utilizando uma política de *round-robin* para atribuir o *mutex* global aos sistemas operativos e estes posteriormente, atribuirão o *mutex* a apenas uma das suas tarefas bloqueadas.

Todas as funções da API foram implementadas sobre um mecanismo de partilha de dados e notificação entre os sistemas operativos. Este mecanismo é constituído por estruturas de dados em memória partilhada, uma por cada sistema operativo, compostas pelos seguintes campos:

- `os_id`: identificador único que identifica o sistema operativo dono da estrutura de dados.
- `cpu_id`: identificador do CPU responsável por atender as IPIs referentes à sincronização.
- `ios_ipi`: número da IPI utilizada pelo sistema operativo para a sincronização.
- `key`: utilizada para libertar os sistemas operativos que se encontram à espera que outro conclua uma atividade.
- `key_lock`: contador que permite indicar ao sistema operativo quantos sistemas operativos se encontram bloqueados à sua espera para retomarem a sua execução.
- `spinlock`: utilizado para garantir o acesso exclusivo à estrutura de dados.
- `signal_type`: identifica o tipo de sinal enviado ao sistema operativo.
- `attribute`: campo por onde os sistemas operativos podem enviar dados.

Para além das estruturas de dados partilhadas, este também possui um serviço de atendimento à IPI de sincronização, uma por cada sistema operativo, responsável por descodificar o tipo de pedido e chamar a função disponibilizada pelo sistema operativo para atender o pedido recebido.

Assim, para que um sistema operativo possa enviar um sinal referente à sincronização, para outro sistema operativo, este tem de:

1. Adquirir acesso exclusivo à estrutura de dados partilhada do sistema operativo, com o qual pretende comunicar.
2. Preencher os campos `signal_type` e `attribute`.
3. Enviar uma IPI para o CPU especificado no campo `cpu_id`.

Por sua vez, o sistema operativo que recebe o sinal vai:

1. Executar o serviço de atendimento à IPI de sincronização.
2. Ler os campos `signal_type` e `attribute`, da sua estrutura de dados partilhada.
3. Libertar o acesso à sua estrutura de dados partilhada.
4. Executar a função, disponibilizada pelo sistema operativo, referente ao pedido recebido.

Contudo, o mecanismo de notificação necessita que o sistema operativo disponibilize as funções, apresentadas na tabela 4.1, que permitem a instalação e ativação da IPI referente à sincronização, e o envio de IPIs para outros sistemas operativos. Estas têm de ser implementadas pelo sistema operativo, pois estas funcionalidades variam entre os sistemas operativos.

Tabela 4.1: API específica do sistema operativo para gestão das interrupções

Função	**Descrição**
`GLOBAL_INSTALL_ISR`	Função disponibilizada pelo sistema operativo para instalar interrupções.
`GLOBAL_SEND_IPI`	Função disponibilizada pelo sistema operativo para enviar IPIs.

Holding_Pen **Global**

O mecanismo *holding_pen* global tem como principal objetivo providenciar um meio de sincronização inicial, antes da API de sincronização global se encontrar disponível. Nesse sentido, este não utiliza o mecanismo de notificação implementado, e apenas utiliza os campos `key` e `key_lock`, das estruturas partilhadas. Assim, este mecanismo pode ser utilizado para sincronizar a inicialização dos sistemas operativos, sendo esta muitas vezes requerida devido à partilha de recursos.

Na tabela 4.2 estão listadas as funções implementadas, juntamente com a descrição das suas funcionalidades, que constituem o mecanismo *holding_pen* global.

Tabela 4.2: API *holding_pen* global

Função	Descrição
`global_holding_pen`	Suspende a execução do sistema operativo até ser libertado o `holding_pen`.
`global_realease_pen`	Liberta todos os sistemas operativos presos no `holding_pen`.

Mutex Global

Os *mutexs* globais foram implementados de forma a possibilitar uma fácil gestão de acessos concorrentes a recursos partilhados entre sistemas operativos. Ao contrário do que acontece na API da POSIX os *mutexs* globais têm de ser primeiro criados com recurso à função `get_global_mutex`. Cada *mutex* global é identificado, por todos os sistemas operativos, através de um identificador único, que é atribuído a estes na sua criação. Sempre que se tenta criar um *mutex* com um identificador já existente apenas é retornado um apontador para este. As funções relativas à utilização dos *mutexs* globais estão listadas na tabela 4.3, juntamente com uma descrição.

Tabela 4.3: API *mutex* global

Função	Descrição
`get_global_mutex`	Cria um *mutex* global.
`global_mutex_init`	Inicializa o *mutex* global especificado. Tem de ser invocada para o *mutex* poder ser utilizado.
`global_mutex_destroy`	Destrói o *mutex* global especificado, caso este não esteja a ser utilizado.
`global_mutex_lock`	Adquire o *mutex* global especificado. Se o *mutex* já se encontrar adquirido suspende a tarefa.
`global_mutex_trylock`	Tenta adquirir o *mutex* global especificado. Caso este já se encontre adquirido retorna um erro.
`global_mutex_unlock`	Liberta o *mutex* global especificado.

Além disso, como a gestão interna dos sistemas operativos da suspensão e libertação das tarefas é independente da API, cada sistema operativo tem de implementar os métodos apresentados na tabela 4.4.

Tabela 4.4: API específica do sistema operativo para gestão interna dos *mutexs* globais

Função	Descrição
`GLOBAL_MUTEX_LOCK`	Função específica do sistema operativo, responsável por gerir a suspensão das tarefas.
`GLOBAL_MUTEX_UNLOCK`	Função específica do sistema operativo, responsável por resumir as tarefas.

Conditonal variable Global

As *conditonal variables* globais disponibilizam um meio de sincronização entre tarefas, que se encontram em sistemas operativos diferentes. Tal como acontece nos *mutexs* globais, estas também são identificadas por um identificador único, e além disso, como acontece na API da POSIX, as *conditonal variables* globais têm de ser associadas com um *mutex* global. Na tabela 4.5 é apresentada a lista de funções disponibilizadas, para a utilização das *conditonal variables*.

Tabela 4.5: API *conditional variable* global

Função	Descrição
`get_global_cond`	Cria uma *conditional variable* global.
`global_cond_init`	Inicializa a *conditional variable* global especificada. Tem de ser invocada para a *conditional variable* poder ser utilizada.
`global_cond_destroy`	Destrói a *conditional variable* global especificada, caso esta não esteja a ser utilizada.
`global_cond_signal`	Sinaliza a *conditional variable*. Liberta uma das tarefas em espera na *conditional variable*.
`global_cond_broadcast`	Liberta todas as tarefas em espera na *conditional variable*.
`global_cond_wait`	Coloca a tarefa em espera na *conditional variable* global, até esta ser sinalizada.

Na tabela 4.6 estão listadas as funções associadas com as *conditonal variables* globais que cada sistema operativo deve implementar de forma a gerir a suspensão e libertação das suas tarefas.

Tabela 4.6: API específica do sistema operativo para gestão interna das *conditonal variables*

Função	Descrição
`GLOBAL_COND_SIGNAL`	Função específica do sistema operativo, responsável por libertar uma tarefa em espera numa *conditonal variable* global.
`GLOBAL_COND_BROADCATS`	Função específica do sistema operativo, responsável por libertar todas as tarefas em espera numa *conditonal variable* global.
`GLOBAL_COND_WAIT`	Função específica do sistema operativo, responsável por gerir a suspensão das tarefas em espera numa *conditonal variable*.

Semaphore Global

Os *semaphores* globais podem ser utilizados para proteger um recurso partilhado de acesso concorrente, no caso de apenas permitirem um acesso, ou para sincronizar a execução de tarefas. Como os outros mecanismos de sincronização globais mencionados, os *semaphores* globais também são identificados no sistema por um identificador único. As funções disponibilizadas para a utilização dos *semaphores* globais estão apresentadas na tabela 4.7, juntamente com um descrição.

Tabela 4.7: API *semaphore* global

Função	Descrição
`get_global_sem`	Cria um *semaphore* global.
`global_sem_init`	Inicializa o *semaphore* global especificado. Tem de ser invocada para o *semaphore* poder ser utilizado.
`global_sem_destroy`	Destrói o *semaphore* global especificado, caso este não esteja a ser utilizado.
`global_sem_post`	Incrementa o valor do contador do *semaphore* global. Caso exista tarefas bloqueadas neste liberta uma.
`global_sem_wait`	Decrementa o valor do contador do *semaphore* global. Caso este seja maior ou igual a zero é dado acesso à tarefa.
`global_sem_getvalue`	Devolve o valor atual do contador do *semaphore* global.

Na tabela 4.8 estão listadas as funções, específicas de cada sistema operativo, associadas com os *semaphores* globais, responsáveis por resumir e suspender as tarefas.

Tabela 4.8: API específica do sistema operativo para gestão interna dos *semaphores* globais

Função	Descrição
`GLOBAL_SEM_LOCK`	Função específica do sistema operativo, responsável por gerir a suspensão das tarefas.
`GLOBAL_SEM_UNLOCK`	Função específica do sistema operativo, responsável por resumir as tarefas.

4.3.2 Comunicação entre Sistemas Operativos

A implementação de um mecanismo de comunicação entre diferentes sistemas operativos tem inúmeras vantagens associadas. Por um lado possibilita que dados processados por uma tarefa num sistema operativo possam ser partilhados com outras tarefas, independentemente se estas se encontram ou não no mesmo sistema operativo. Facilitando assim, a partição das aplicações de forma retirar maior proveito das caraterísticas oferecidas por diferentes sistemas operativos.

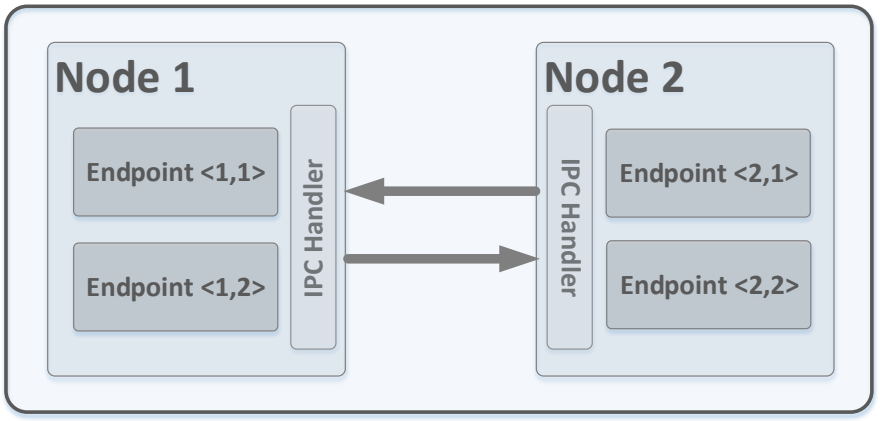

Figura 4.10: Topologia da comunicação

O mecanismo de comunicação implementado foi baseado no MCAPI. O MCAPI apenas especifica a API e não a implementação nem a camada de transporte utilizada, tornando necessário que sempre que se pretenda utilizar a API do MCAPI seja necessário implementar todo o mecanismo de comunicação, levando a que a integração deste seja uma tarefa mais morosa e complexa, mas no entanto permite que a implementação deste vá de encontro às necessidades do sistema. No entanto, a comunicação implementada especifica a API, a camada de transporte, e ainda disponibiliza toda a implementação da comunicação. Assim, como esta tem o propósito de poder ser facilmente integrada nos sistemas operativos, sem

ser necessário a realização de alterações, esta seria inflexível, pois teria sempre o mesmo funcionamento. Por isso, de forma a contornar este inconveniente, foram adicionados modos de configuração para poder ser possível configurar o comportamento da comunicação sem ser necessário alterar a implementação desta. Dessa forma, continua a não ser necessário implementar todo o mecanismo de comunicação, ao contrário do que acontece no MCAPI, sem se abdicar da configurabilidade da comunicação, para diferentes casos de uso.

Nesse sentido, a influência do MCAPI está maioritariamente presente na API disponibilizada e na topologia utilizada. O mecanismo de comunicação implementado, ao contrário do que acontece no MCAPI, que possibilita o envio de pacotes de dados, valores escalares e mensagens, apenas possibilita a passagem de mensagens. Contudo, a comunicação implementada permite, que juntamente com as mensagens, seja enviada informação adicional, de forma a possibilitar a implementação de mecanismos mais complexos sobre a passagem de mensagens. Além disso, como mencionado anteriormente, a comunicação também pode ser configurada, durante a compilação, em diferentes modos de funcionamento, de forma a que esta apresente um funcionamento mais vantajoso para o sistema.

O meio utilizado na implementação da camada de transporte foi a memória partilhada. Pois, como analisado na secção 3.1.5, o ARM Cortex-A9 permite a configuração de regiões de memória como sendo *strongly orderd*, onde é garantido que todos os acessos, a essas regiões de memória, são visíveis imediatamente por todas as unidades de processamento. Nesse sentido, todas as estruturas de dados, necessárias à implementação do mecanismo de comunicação, foram instanciadas num bloco de memória *strongly orderd* comum a todos os sistemas operativos.

A comunicação implementada é baseada nas abstrações *node*, *endpoint* e *handler*, conforme se pode observar na figura 4.10, e permite uma comunicação bidirecional em paralelo entre diferentes entidades de comunicação. Seguidamente, são apresentados os principais conceitos utilizados e o funcionamento do mecanismo implementado.

Nodes

Os *nodes* são uma abstração lógica, utilizados para mapear os sistemas operativos. Estes são definidos em tempo de compilação e são sempre únicos pois, cada *node* é atribuído a apenas um sistema operativo. Além disso, cada *node* possui

uma interface, acessível por todos os *nodes* do sistema, onde são disponibilizadas informações essenciais para que as entidades de comunicação dos *nodes* possam comunicar corretamente entre si. De seguida é apresentada a interface de cada *node*, e a respetiva descrição de cada campo constituinte desta:

- `node_id`: identificador único do *node*, atribuído durante a compilação.
- `cpu_handler`: especifica qual a unidade de processamento para onde a IPI de comunicação deve ser enviada, caso o *node* a utilize.
- `mode`: especifica qual o modo de operação que o *node* utiliza internamente.
- `info`: indica qual a número da IPI de comunicação utilizada pelo *node*.
- `node_state`: indica o estado do *node*, por exemplo se se encontra inicializado ou não.
- `endpoint_list`: lista com todos os *endpoints* que o *node* possui.
- `request_lock`: gere os acessos concorrentes aos blocos de comunicação.
- `request_count`: contém o número de pedidos de comunicação recebidos.
- `request_first`: fila de pedidos de comunicação recebidos. Aponta para o pedido mais antigo.
- `request_last`: fila de pedidos de comunicação recebidos. Aponta para o pedido mais recente.
- `request_queue`: lista com todos os pedidos de comunicação disponíveis disponíveis no *node*.

É importante salientar que as mensagens são enviadas dentro dos pedidos de comunicação. Nesse sentido, para que uma mensagem possa ser enviada primeiro é necessário adquirir um pedido de comunicação e posteriormente devolver este ao *node* com a mensagem, utilizando o mecanismo especificado pelo modo de configuração do *node*.

Endpoints

Os *endpoints* são entidades de comunicação pertencentes aos *nodes*. Estes são criados em tempo de execução e permitem o envio e receção de mensagens. Cada *node* pode conter vários *endpoints* e estes apresentam uma identificação global única,

atribuída na criação de um novo *endpoint*, através da combinação do identificador do *node*, a que este pertence, mais o identificador do porto de comunicação atribuído ao *endpoint* `<node_id,port_id>`. A referência de um *endpoint* pode ser obtida por qualquer *node*, que posteriormente pode utilizar a referencia do *endpoint* obtida para enviar-lhe mensagens.

Handlers

Ao contrário dos dois conceitos apresentados anteriormente, que foram baseados na topologia do MCAPI, os *handlers* são um novo conceito introduzido nesta implementação. Os *hanlders* são blocos de código responsáveis por averiguar a receção de novas mensagens, e reencaminhar estas para o sistema operativo. Contudo, a utilização de um *handler* por parte de um *node* está relacionada com a configuração escolhida para este, durante a compilação, uma vez que nem todos os modos de configuração necessitam de um *handler*, como será posteriormente explicado.

O propósito dos *handlers* é o de permitir que sejam os sistemas operativos a gerir o armazenamento das mensagens recebidas, possibilitando assim que os sistemas operativos possam implementar mecanismos mais sofisticados (p.e., suporte a prioridades, suspensão das tarefas em espera de novas mensagens, notificação das tarefas da receção de uma nova mensagem), de forma transparente às tarefas, sobre o mecanismo de comunicação implementado. Além disso, estes também permitem que os sistemas operativos possam configurar o funcionamento da comunicação de forma a cumprir requisitos que estes possam ter.

Suporte do sistema operativo

Tendo em consideração que a API de comunicação implementada é independente do sistema operativo, torna necessário que o sistema operativo implemente as funcionalidades, específicas deste, que a API de comunicação necessita. O suporte requerido ao sistema operativo pela API de comunicação varia consoante o modo de configuração utilizado. No entanto, independentemente do modo, o sistema operativo tem sempre de implementar as funções que permitem a gestão de acessos concorrentes, mais uma função que permita o envio de IPIs para os restantes sistemas operativos. Essas funções são apresentadas na tabela 4.9 juntamente com as suas descrições.

Tabela 4.9: Funções implementadas pelo sistema operativo

Função	Descrição
IPC_LOCK_INIT	Inicializa o mecanismo de exclusão mútua do *node*.
IPC_LOCK_DESTROY	Destrói o mecanismo de exclusão mútua do *node*.
IPC_LOCK_NODE	Bloqueia o acesso às estruturas internas do *node*.
IPC_UNLOCK_NODE	Liberta o acesso às estruturas internas do *node*.
IPC_SEND_IPI	Envia uma IPI.

No caso da comunicação ser configurada num dos modos que utiliza um *handler*, o sistema operativo tem de implementar as funções relativas à gestão das mensagens recebidas. Além dessas funções, o sistema operativo também pode implementar funções que permitem que este tenha conhecimento dos *endpoints* existentes dentro do *node*. Contudo, essas funções não têm influência na gestão dos *endpoints*, que são sempre geridos diretamente pela API disponibilizada às aplicações. Na tabela 4.10 estão listadas as funções requeridas pelos *handlers* e a respetiva descrição.

Tabela 4.10: Funções requeridas nos modos que usam *handlers*

Função	Descrição
IPC_CREATE_ENDPOINT	Sinaliza o sistema operativo da criação de um novo *endpoint*.
IPC_DELETE_ENDPOINT	Sinaliza o sistema operativo da remoção de um novo *endpoint*.
IPC_RECV_MSG	Função responsável por armazenar as mensagens recebidas.
IPC_READ_MSG	Função de leitura das mensagens recebidas.
IPC_AVAILABLE_MSG	Retorna o numero de mensagens pendentes que um *endpoint* possui.

Para além das funções apresentadas, o sistema operativo pode ainda ter de disponibilizar mais funções. Essas funções são específicas de cada modo, e serão analisadas juntamente com o modo de configuração a que pertencem.

ISR handler mode

O modo *ISR handler* é um dos modos que utiliza um *handler*, para atender as mensagens recebidas pelo *node*. O *handler* neste caso será executado como sendo um serviço de atendimento à interrupção, logo é necessário que o sistema operativo disponibilize os mecanismos, apresentados na tabela 4.11, de forma a permitir a

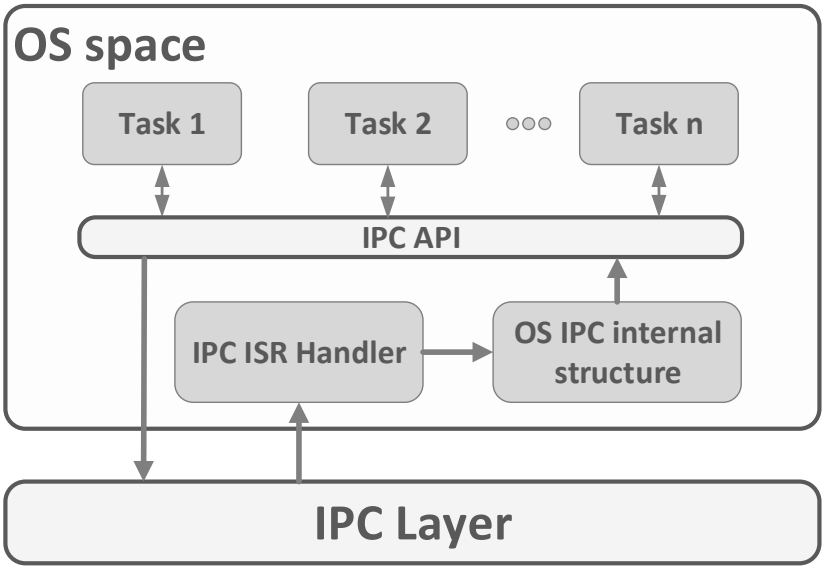

Figura 4.11: *ISR handler mode* funcionamento

instalação e desinstalação do *handler*. O funcionamento deste modo pode ser observado na figura 4.11 e segue as seguintes passos:

1. O *endpoint* que pretende enviar a mensagem envia uma IPI para o *node* que possui o *endpoint*, com o qual pretende comunicar.
2. O *handler* do *node* é executado quando o sistema operativo recebe a IPI.
3. O *hanlder* lê a mensagem e reencaminha esta para o sistema operativo.
4. O sistema operativo armazena a mensagem de acordo com as suas especificações.
5. Sempre que um *endpoint* pretende ler uma mensagem este utiliza a API de comunicação, que por sua vez irá pedir ao sistema operativo por uma mensagem destinada ao *endpoint*.

Tabela 4.11: Funções requeridas pelo modo *ISR handler*

Função	**Descrição**
`IPC_INSTALL_ISR`	Instala o serviço de atendimento à interrupção que executa o *handler*.
`IPC_UNINSTALL_ISR`	Desinstala o serviço de atendimento à interrupção referente ao *handler*.

Thread handler mode

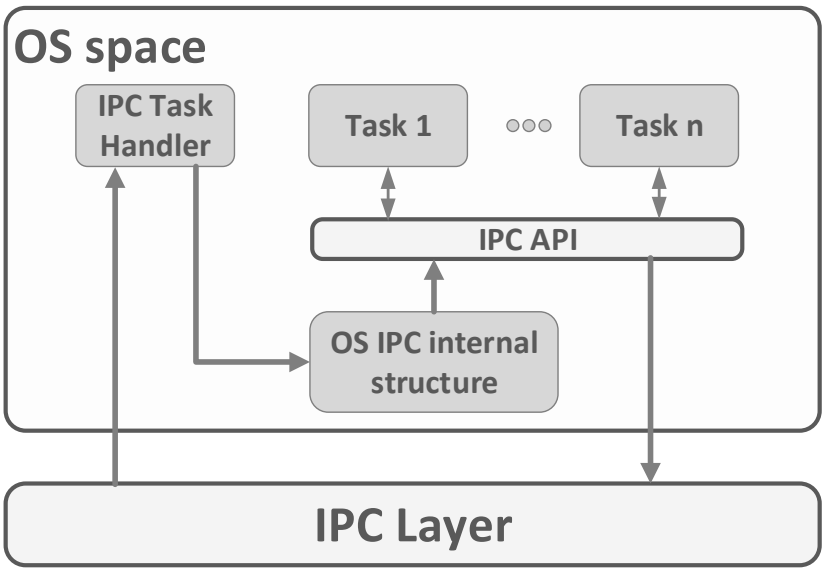

Figura 4.12: *Thread handler mode* funcionamento

O modo *thread handler*, tal como acontece no modo *ISR handler*, também utiliza um *handler*. No entanto, ao contrário do modo *ISR handler*, o *handler* neste modo é executado como sendo uma tarefa pertencente ao sistema operativo. Logo, a execução do *handler* encontra-se sujeita às políticas de escalonamento do sistema operativo. Além disso, o sistema operativo tem de providenciar as funções listadas na tabela 4.12, de forma a permitir a criação e desinstalação da tarefa do *hanlder*. O funcionamento deste modo pode ser observado na figura 4.12 e segue os seguintes passos:

1. O *endpoint* que pretende enviar a mensagem incrementa, de forma atómica, o campo `request_count` da estrutura de dados partilhada do *node* que possui o *endpoint*, com o qual pretende comunicar.

2. A tarefa do *handler* encontra-se em *polling* no campo `request_count` do *node*, e sempre que este é maior do que zero (existe mensagens novas pendentes) lê uma nova mensagem e reencaminha-a para o sistema operativo.

3. O sistema operativo armazena a mensagem de acordo com as suas especificações.

4. Sempre que um *endpoint* pretende ler uma mensagem este utiliza a API de comunicação, que por sua vez irá pedir ao sistema operativo por uma

mensagem destinada ao *endpoint*.

Para além do funcionamento base explicado, o modo *thread handler* pode ser configurado para utilizar a API de sincronização global 4.3.1. Esta configuração trás uma grande vantagem, que reside no facto de a tarefa do *handler* não ter de ficar em *polling* à espera de novas mensagens, desperdiçando tempo de processamento, e passa a poder ser suspendida através da API de sincronização global, e apenas resumida quando existirem mensagens novas para serem atendidas.

No entanto, para que este modo possa utilizar esta configuração, todos os sistemas operativos que utilizem a API de comunicação têm de utilizar também a API de sincronização global. Isto deve-se ao facto da sinalização, da receção de novas mensagens, ser efetuada através dos mecanismos da sincronização global, sendo portanto necessário que os *endpoints*, que pretendam enviar mensagens para um *node* com esta configuração, tenham acesso aos mecanismos da sincronização global.

Tabela 4.12: Funções requeridas pelo modo *thread handler*

Função	**Descrição**
`IPC_CREATE_THREAD`	Cria a tarefa que executa o *handler* .
`IPC_UNINSTALL_THREAD`	Desinstala a tarefa referente ao *handler*.

No handler mode

O modo *no handler* é o único modo de configuração que não utiliza um *handler*. Nesse sentido, este também não necessita de suporte adicional, por parte do sistema operativo, para além das funções listadas na tabela 4.9. Este utiliza as estruturas internas do mecanismo de comunicação, para armazenar as mensagens recebidas seguindo uma política FIFO. Para além disso, como se pode observar na figura 4.13, o sistema operativo não intervém no processo de comunicação.

O facto de não requerer suporte adicional torna a sua integração no sistema operativo mais simples e rápida. Contudo, torna também a comunicação menos flexível e limitada ao nível de armazenamento de mensagens, pois fica limitada aos *buffers* de receção da API de comunicação. Este último inconveniente pode ser atenuado configurando durante a compilação a comunicação para utilizar *buffers* para receção de mensagens maiores. Esta configuração implica também que as tarefas que

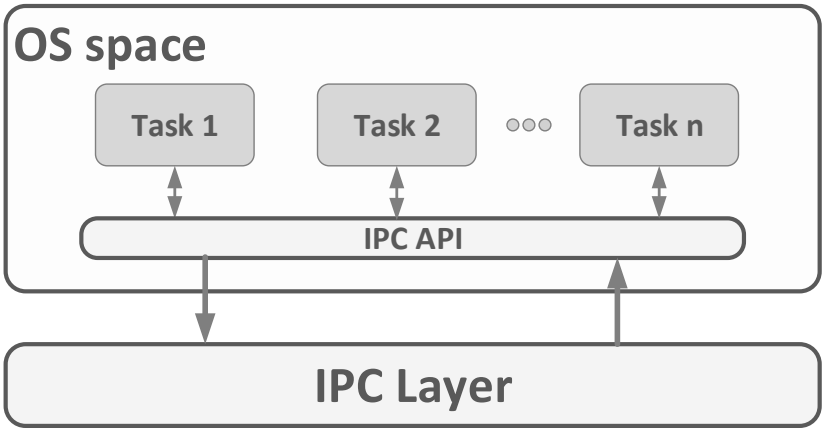

Figura 4.13: *No handler mode* funcionamento

possuam *endpoints*, que se encontrem à espera de mensagens, fiquem em *polling* até receberem uma nova mensagem.

No entanto, tal como acontece no modo *thread handler*, este modo também pode ser configurado de forma a utilizar a API de sincronização global. Dessa forma, é possível aumentar a flexibilidade da comunicação no sentido que as tarefas que possuem *endpoints*, à espera de mensagens, podem ser suspendidas e novamente retomadas quando o *endpoint* recebe uma nova mensagem

API de comunicação implementada

Como se pode constatar da análise da comunicação entre sistemas operativos implementada, esta permite um elevado nível de configuração. No entanto, esta configurabilidade é totalmente transparente na perspetiva das aplicações, que independentemente da configuração da comunicação utilizam sempre a mesma API (tabela 4.13). Além disso, a comunicação foi implementada de forma a permitir que todos os *nodes* do sistemas possam comunicar entre si, mesmo tendo configurações diferentes.

Os mecanismos de comunicação foram implementados de forma a permitir a comunicação bidirecional em simultâneo entre todos os *nodes*. Nesse sentido, é possível ter mais do que um *endpoint* a enviar mensagens para o mesmo *endpoint* destinatário em simultâneo, e ainda ter este último a enviar mensagens para qualquer outro *endpoint*. Isto é conseguido utilizando métodos atómicos, na aquisição de

Tabela 4.13: API de comunicação entre sistemas operativos

Função	Descrição
`ipc_initialize`	Inicializa a comunicação entre sistemas operativos no *node* local.
`ipc_finalize`	Termina a comunicação entre sistemas operativos no *node* local.
`node_id_get`	Retorna o identificador associado ao *node* local.
`endpoint_create`	Cria um novo *endpoint* no *node* local.
`endpoint_delete`	Apaga o *endpoint* especificado do *node* local.
`endpoint_get`	Devolve o identificador do *endpoint* associado com o *node_id* e *port_id* especificados.
`msg_send`	Envia uma mensagem de um *endpoint* local para o *endpoint* especificado.
`msg_recv`	Rotina para receber mensagens do *endpoint* local especificado.
`msg_available`	Verifica se há mensagens pendentes no *endpoint* local especificado.

pedidos de comunicação, já que o algoritmo de envio de mensagens encontra-se dividido em três etapas distintas:

1. Aquisição de um pedido de comunicação ao *node* dono do *endpoint* destinatário (efetuado utilizando métodos atómicos em vez da utilização de *locks*).

2. Preenchimento do pedido de comunicação com a mensagem e outros dados requeridos (e.g. identificador do *endpoint* destinatário, tamanho da mensagem).

3. Notificação do *node* dono do *endpoint* destinatário, caso este use *handler*; ou do *endpoint* destinatário, caso este não use *handler*, através do método indicado pelo modo em que o *node* se encontra configurado.

Como se pode verificar na tabela 4.13, a API apenas permite o envio de mensagens sem ser necessário conectar os *endpoints*, para que estes possam comunicar entre si. Juntamente com as mensagens é enviado o tamanho desta e mais um campo adicional, denominado de `code`, que pode ser utilizado para enviar informações adicionais, caso seja implementado algum sistema mais complexo sobre a API. Além disso, também é necessário especificar o *endpoint* a que a mensagem se destina e o *endpoint* que a envia.

O envio de mensagens é sempre *non-blocking*, isto é, a função de envio não espera pela confirmação da receção da mensagem e retorna de imediato. No que diz

respeito à função de receção, na implementação base (modo *no handler*) sempre que um *endpoint* tenta ler uma nova mensagem sem que haja mensagens disponíveis este fica bloqueado até receber uma mensagem. Caso a implementação use um modo com *handler* fica ao critério do sistema operativo definir o comportamento.

4.3.3 Hybrid multiprocessing

O *hybrid multiprocessing* surge como uma forma de conjugar, apenas num sistema, as vantagens que as abordagens SMP e AMP oferecem. Ao contrário do que acontece por exemplo, na arquitetura BMP, que possibilitam a atribuição de tarefas a unidades de processamento, como forma de preservar até certo ponto as caraterísticas originais do sistema operativo (ambiente de execução *singlecore*), bem como garantir o correto comportamento de aplicações *legacy*. O HMP oferece os dois ambientes de execução (*multicore* e *singlecore*) simultaneamente, auxiliados por mecanismos de comunicação e sincronização. Dessa forma, é possível manter a compatibilidade com as aplicações *legacy* e as caraterísticas originais do sistema operativo sem abdicar das vantagens providenciadas pelos processadores *multicore*, tendo um *footprint* de memória menor quando comparado com um sistema AMP.

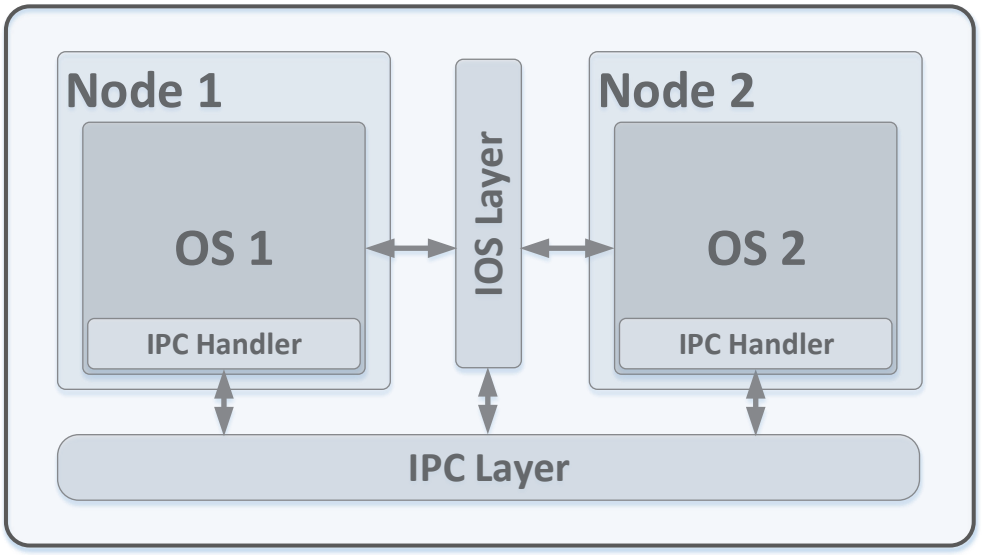

Figura 4.14: Sistema HMP desenvolvido

Os principais requisitos de um sistema HMP é a capacidade dos sistemas operativos constituintes do sistema serem capazes de coordenar a sua execução (e.g. acesso a recursos partilhados) e que as aplicações sejam capazes de comunicar e

sincronizar as suas atividades entre si independentemente do sistema operativo a que pertençam. Nesse sentido, num sistema HMP é vital a existência de mecanismos de sincronismo e comunicação entre os sistemas operativos. Assim sendo, a versão HMP do ARM *microkernel* desenvolvida nesta dissertação utiliza os mecanismos de comunicação e sincronização apresentados previamente, de forma a cumprir esses requisitos. Na figura 4.14 pode-se observar como estas APIs são utilizadas para interligar dois sistemas operativos, num sistema HMP.

Inicialização do ARM *microkernel* HMP

No ARM *microkernel* HMP o processo de inicialização tem de ter em consideração, o facto de os sistemas operativos terem a necessidade de coordenar os seus processos de inicialização individuais. Isto deve-se ao facto de existirem recursos partilhados que apenas devem ser inicializados por um dos sistemas operativos, tal como acontecia na versão AMP. Nesse sentido, durante o processo de inicialização, como se pode observar na figura 4.15, a versão SMP será a responsável por realizar essas inicializações.

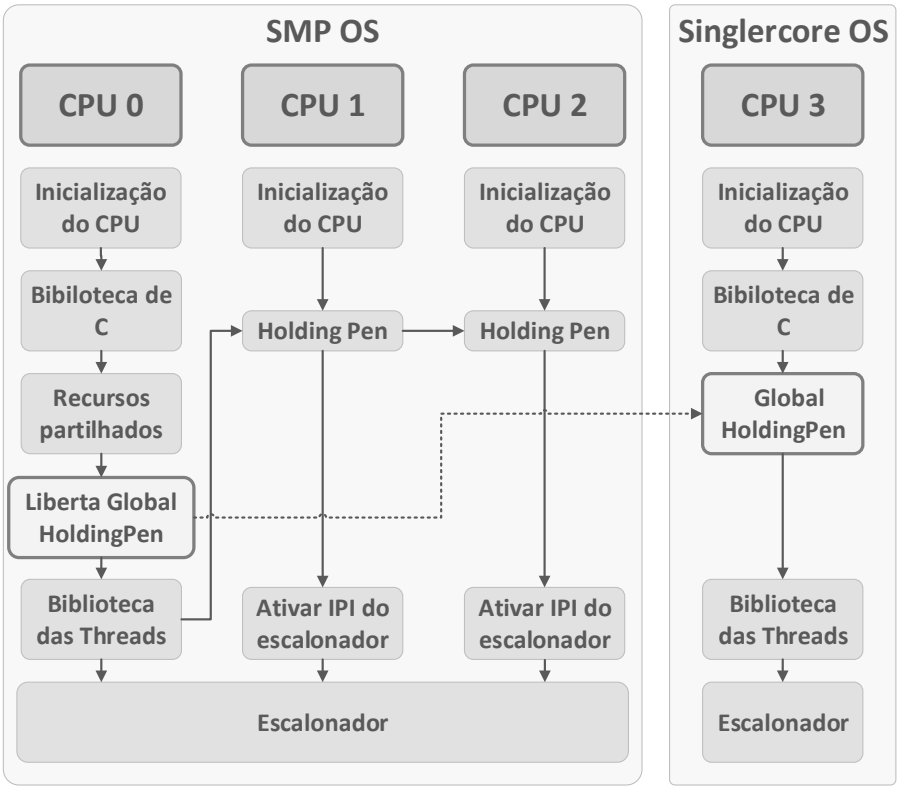

Figura 4.15: Inicialização do ARM *microkernel* HMP

Os processos de inicialização, de ambos os sistemas operativos, praticamente não sofreram alterações, comparativamente com as suas implementações originais. Tendo sido apenas necessário adicionar a ambos o mecanismo do `Holding_Pen` global, para permitir a sincronização da inicializações de ambos os sistemas operativos. Pois neste caso, a versão *singlecore* para concluir a sua inicialização precisa que recursos de *hardware* e também recursos de *software*, como as APIs de comunicação e sincronização globais, se encontrem inicializados. Assim, com a utilização do `Holding_Pen` global, a versão *singlecore* irá ficar bloqueada à espera que a versão SMP finalize todas as inicializações dos recursos partilhados, e liberte o `Holding_Pen` global.

Configurabilidade do ARM *microkernel* HMP

O ARM *microkernel* HMP implementado pode ser configurado em diferentes arquiteturas de multi-processamento. Nesse sentido, os códigos das versões SMP e AMP do *microkernel* foram combinados num só através do uso de compilação condicional. Dessa forma, é possível alterar a configuração do *microkernel* sem ser necessário efetuar alterações ao código deste, pois apenas é necessário configurar um conjunto de *macros* responsáveis por alterar a configuração do *microkernel*. As configurações possíveis para o *microkernel* HMP são as seguintes:

1. ***Singlecore***: apenas uma cópia do sistema operativo a correr numa unidade de processamento.

2. **AMP**: pode ser *dualcore*, *tricore* e *quadcore*.

3. **SMP**: pode ser *dualcore*, *tricore* e *quadcore*.

4. **HMP**: SMP *tricore* mais *singlecore* (exemplo apresentado na figura 4.15), SMP *dualcore* mais *singlecore* ou SMP *dualcore* mais AMP *dualcore*.

Apesar de na corrente dissertação o número máximo de unidades de processamento, por processador, utilizadas ter sido quatro, o ARM *microkernel* HMP encontra-se preparado para ser executado num número mais elevado de unidades de processamento.

4.3.4 Hybrid and heterogeneous multiprocessing

A arquitetura *hybrid and heterogeneous multiprocessing* leva a abordagem do HMP mais longe. Pois, ao invés dos sistemas operativos constituintes do sistema serem todos variantes do mesmo, no H^{2}MP os sistemas operativos podem ser diferentes uns dos outros, daí a designação de sistema heterogéneo.

Um sistema heterogéneo permite que as diferentes tarefas constituintes de uma aplicação sejam distribuídas pelos diferentes sistemas operativos, com o intuito de tirar proveito dos requisitos funcionais destes. Um exemplo seria um RTOS, responsável por executar os serviços que necessitam de tempo-real e determinismo, em paralelo com um sistema operativo de propósito geral, responsável por disponibilizar um interface gráfica ao utilizador. Contudo, neste modelo, tal como acontecia no HMP, torna-se necessário a existência de mecanismo de comunicação e sincronização entre os sistemas operativos. Além disso, o facto de serem sistemas operativos diferentes pode tornar a gestão dos recursos partilhados ainda mais complexa.

O H^{2}MP implementado na presente dissertação conjuga no mesmo sistema o ARM *microkernel* HMP e o FreeRTOS. Nesse sentido, o FreeRTOS foi expandido de forma a integrar as APIs de comunicação e sincronização desenvolvidas, e a fazer uso destas para coordenar a sua inicialização e acesso a recursos partilhados, com o ARM *microkernel* HMP.

Integração do FreeRTOS no H^{2}MP

A integração do FreeRTOS no sistema H^{2}MP foi efetuada por etapas. Na fase inicial foi analisado o código do FreeRTOS com o objetivo de compreender o funcionamento deste. Esta análise incidiu essencialmente em três pontos (i) processo de inicialização; (ii) recursos de hardware utilizados; e (iii) mecanismos de suspensão e resumo de tarefas.

Após a conclusão da análise do FreeRTOS, transitou-se para a integração das APIs de comunicação e sincronização neste. Contudo, esta etapa foi simplificada pelo facto de estas APIs terem sido desenvolvidas com o propósito de serem facilmente integradas nos sistemas operativos. Assim, não foi necessário realizar alterações ao código do FreeRTOS, para além da adição do suporte requerido pelas APIs.

Por fim, na última etapa, o processo de inicialização foi alterado com o objetivo de

adicionar o suporte necessário para que este possa ser sincronizado com o processo de inicialização do ARM *microkernel* HMP. Pois, este último é o responsável por lançar o FreeRTOS e inicializar todos os recursos partilhados. Além disso, os acessos aos recursos partilhados foram protegidos, com os mecanismos de sincronização adequados.

Sistema H^{2}MP: ARM *microkernel* HMP mais FreeRTOS

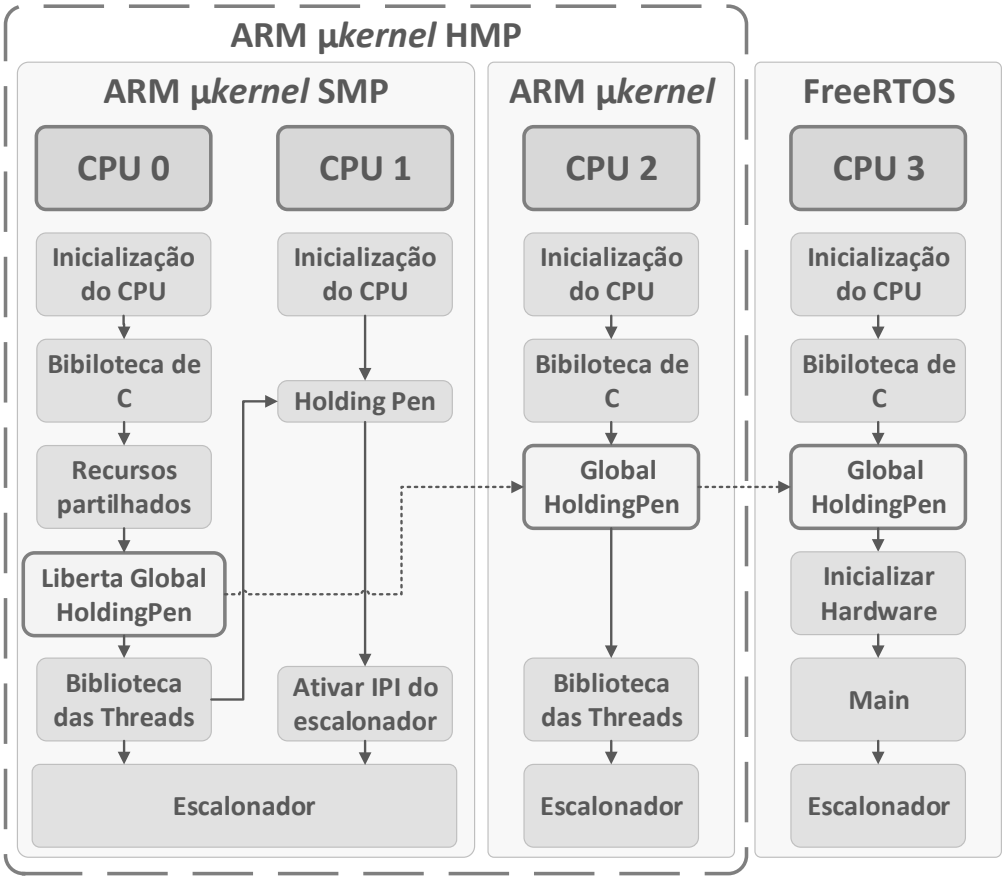

Figura 4.16: Inicialização do sistema H^{2}MP

Como mencionado anteriormente, os sistemas operativos constituintes do H^{2}MP implementado são o ARM *microkernel* HMP e o FreeRTOS, que por sua vez se encontram interligados pelas APIs de comunicação e sincronização. Tendo em consideração o facto de no arranque do sistema H^{2}MP ser necessário que um dos sistemas operativos seja responsável por arrancar os restantes sistemas operativos e inicializar todos os recursos partilhados, foi definido que essa responsabilidade seria atribuída ao ARM *microkernel* HMP. Como este já se encontra preparado

para executar essas funções, como foi analisado na secção 4.3.3, não foi requerido a realização de alterações ao seu código. Pois, desde que todos os sistemas operativos constituintes do sistema usem as mesmas APIs de sincronização e comunicação, não existe diferença entre o sistema ser homogéneo ou heterogéneo.

Na figura 4.16 é exposto um exemplo do processo de inicialização do sistema H^{2}MP. O H^{2}MP apresentado na figura é constituído por três sistemas operativos, sendo que dois destes fazem parte do ARM *microkernel* HMP, e os respetivos processos de inicialização. O mecanismo `Holding_Pen` global, disponibilizado pela API de sincronização, é utilizado não só para sincronizar o arranque dos sistemas operativos pertencentes ao ARM *microkernel* HMP mas também para o do FreeRTOS. Dessa forma o FreeRTOS quando executa as rotinas de inicialização do *hardware* que necessita e posteriormente a sua `main`, sabe que todos os recursos partilhados já se encontram inicializados e prontos a serem utilizados. Além disso, a seguir ao `Holding_Pen` global os sistemas operativos já podem utilizar todos os mecanismos providenciados pelas APIs de sincronização e comunicação, de forma a lidar com a partilha de recursos e comunicações necessárias durante a execução destes.

Capítulo 5

Resultados Experimentais

No capítulo anterior foi apresentada a implementação do sistema, começando pelo *porting* do ARM *microkernel* SMP para as plataformas de desenvolvimento Zynq-7000 e VE, seguindo-se o *refactoring* deste para as versões *singlecore*, AMP e HMP. Além disso, foram também apresentadas as implementações das APIs de sincronização e comunicação entre sistemas operativos, que serviram de suporte às arquiteturas AMP e HMP.

Relativamente ao presente capítulo, são apresentados os resultados experimentais dos testes realizados, no emulador ARM Fast Models, para avaliar o desempenho das diferentes versões do *microkernel* e das APIs de sincronização e comunicação. Além disso, são também apresentados os respetivos *footprints* de memória, obtidos com recurso à ferramenta *size* da *toolchain* da GNU. Assim, inicialmente são analisados e comparados os *overheads* de memória das versões implementadas do *microkernel*, e das APIs desenvolvidas. Posteriormente, é também avaliado o desempenho de cada versão do *microkernel*, de forma a estabelecer uma relação entre os custos de memória e o desempenho obtido. Além disso, também são avaliadas as latências das funcionalidades disponibilizadas pelas APIs de sincronização e comunicação, para os casos independentes do suporte do sistema operativo.

5.1 *Footprint* de memória

A ferramenta *size*, disponibilizada pela *toolchain* da GNU, foi a ferramenta utilizada para a obtenção do *footprint* de memória das diferentes configurações. As-

sim, primeiramente é descrita ferramenta *size*, e posteriormente são apresentados os resultados obtidos. As medições foram efetuas com dois propósitos. Avaliar e comparar o *footprint* de memória das diferentes versões implementadas do *microkernel*, e de avaliar o *overhead* de memória introduzido pelas APIs de sincronização e comunicação. Nesta última foi medido o *overhead* de memória introduzido para todas as configurações que esta possibilita.

5.1.1 Ferramenta *Size* da GNU

Como supramencionado a ferramenta *size*, da *toolchain* da GNU, foi a ferramenta escolhida para adquirir o *footprint* de memória. Esta ferramenta recebe como argumentos uma lista de ficheiros objeto, e posteriormente apresenta uma listagem com o tamanho das secções de memória destes. Para cada secção de memória, dos ficheiros objeto, é gerada uma linha de *output* com o tamanho e endereço destas.

5.1.2 *Microkernel*

No capítulo 4 foram apresentadas e analisadas as diferentes versões do ARM *microkernel* implementadas na presente dissertação. Tendo em consideração que cada versão utiliza uma abordagem de multi-processamento diferente, de forma a tirar proveito das vantagens providenciadas pelos processadores *multicore*, foram realizadas medições relativamente ao impacto que cada uma destas tem na utilização de memória, e posteriormente na secção 5.2 o impacto ao nível do desempenho destas.

Singlecore

Tabela 5.1: *Footprint* de memória do ARM *microkernel singlecore*

	.text	**.data**	**.bss**	**Total**
Singlecore	22576	720	3212	26508

Na tabela 5.1 são apresentados os valores obtidos, em *bytes*, relativos ao *footprint* de memória das secções de código (`.text`), varáveis inicializadas (`.data`) e variáveis não inicializadas (`.bss`), da versão *singlecore* do ARM *microkernel*. Estes valores são utilizados como referência de forma a analisar o impacto que o suporte ao ambiente de processamento *multicore*, tem em termos de utilização de memória.

SMP

Como se pode observar na tabela 5.2, onde apresenta o *footprint* de memória para a configuração SMP, esta apresenta um *footprint* ligeiramente superior à versão *singlecore*. Este aumento verificasse principalmente na secção `.bss`, que reflete o facto de na versão SMP ser necessário a existência de estruturas de dados adicionais, de forma a permitir a gestão da execução de todas as unidades de processamento. O aumento verificado na secção `.text` advém maioritariamente dos mecanismos de sincronização utilizados durante o *boot* do *microkernel*.

Tabela 5.2: *Footprint* de memória do ARM *microkernel* SMP

	.text	**.data**	**.bss**	**Total**
SMP	22728	732	3660	27120

AMP

Os resultados obtidos, da medição do *footprint* de memória para a versão AMP *quadcore* do ARM *microkernel*, estão apresentados na tabela 5.3. Esta apresenta em separado os resultados obtidos para o OS primário, *microkernel* responsável pelas inicializações dos recursos partilhados, e das restantes três imagens do *microkernel*. Conforme se pode observar na tabela apresentada, o *microkernel* denominado de primário tem um *footprint* de memória superior às restantes imagens, que por sua vez têm todas o mesmo *footprint* de memória. Isto deve-se ao facto do OS primário ser o único que possui as secções de código responsáveis pelas inicializações dos recursos partilhados.

Tabela 5.3: *Footprint* de memória do ARM *microkernel* AMP

AMP	**.text**	**.data**	**.bss**	**Total**
OS primário	34380	740	3244	38364
OS secundários	33356*3	740*3	3244*3	37340*3
				150384

Além disso, também se verifica um aumento do *footprint* de memória, por imagem do *microkernel*, quando comparado tanto com a versão *singlecore* como com a versão SMP. Este aumento é consequência da inclusão das APIs de comunicação e sincronização, analisadas posteriormente neste capítulo nas secções 5.1.3 5.1.4, respetivamente.

HMP

Como a versão HMP do ARM *microkernel* combina no mesmo sistema as versões SMP e AMP, na tabela 5.4 são apresentados os *footprint* de memória para o *microkernel* SMP *tricore* e para o *microkernel singlecore*. Tal como acontece na versão AMP, as imagens do *microkernel* no HMP também apresentam um aumento do *footprint* de memória, por imagem, devido à inclusão das APIs de comunicação e sincronização.

Tabela 5.4: *Footprint* de memória do ARM *microkernel* HMP (SMP *tricore* mais *singlecore*)

HMP	**.text**	**.data**	**.bss**	**Total**
SMP *tricore*	34884	748	3528	39160
Singlecore	33356	740	3244	37340
				76500

Como se pode constatar da análise dos resultados expostos na tabela 5.4, as diferenças referentes à utilização de memória das duas imagens do *microkernel*, constituintes do sistema HMP, mantêm uma relação semelhante à verificada entre as versões *singlecore* e SMP. Isto deve-se ao facto de as caraterísticas, destas versões, serem preservadas no sistema HMP. Seguidamente na tabela 5.5, são também apresentados os *footprint* de memória para a versão do HMP constituído por um *microkernel* SMP *dualcore* e um *microkernel* AMP *dualcore*.

Tabela 5.5: *Footprint* de memória do ARM *microkernel* HMP (SMP *dualcore* mais AMP *dualcore*)

HMP	**.text**	**.data**	**.bss**	**Total**
SMP *dualcore*	34780	744	3372	38896
AMP *dualcore*	2*33356	2*740	2*3244	2*37340
				113576

Comparando o *footprint* total das duas versões do HMP com as outras duas versões de multi-processamento, SMP e AMP, pode-se constatar que a primeira versão do HMP apresenta uma aumento de *footprint* de memória de 182% relativamente à versão SMP, e um decréscimo de 96.6% relativamente à versão AMP. No caso da segunda versão do HMP pode constatar-se um aumento do *footprint* de memória de 318.8% relativamente à versão SMP, e um decréscimo de 32.4% relativamente à versão AMP.

5.1.3 API de sincronização

Na tabela 5.6 é apresentado o *overhead* de memória introduzido pela API de sincronização. Os valores apresentados não incluem o *footprint* de memória das estruturas instanciadas em memória partilhada, isto porque para além de apenas existir um bloco de memória que possui essas estruturas (independentemente do número de sistemas operativos) a dimensão deste bloco não é definido pois, o tamanho é especificado em tempo de compilação.

Tabela 5.6: *Overhead* de memória da API de sincronização

	.text	**.data**	**.bss**	**Total**
Sincronização	6756	16	0	6772

5.1.4 API de comunicação

Para a API de comunicação foram avaliados todos os modos de configuração que esta permite. Nesse sentido, na tabela 5.7 são apresentados os valores de *overhead* de memória introduzidos pela API, para os três modos em que esta pode ser configurada, quando esta não utiliza as funcionalidades da API de sincronização. Apesar do modo *ISR handler* não utilizar as funcionalidades da API de sincronização, este também precisa de ter o suporte necessário, para comunicar com outro *node*, que utilize a API de sincronização. Logo, o *overhead* introduzido, por todos os modos de configuração da comunicação, varia consoante a utilização da API de sincronização.

Tabela 5.7: *Overhead* de memória da API de comunicação

	.text	**.data**	**.bss**	**Total**
ISR Handler	4776	4	12	4792
Thread Handler	4732	4	12	4748
No Handler	4400	4	12	4416

Na tabela 5.8 são então apresentados os valores do *overhead* de memória introduzido pela API de comunicação, quando esta utiliza a API de sincronização. No entanto, nas duas tabelas apresentadas, o *footprint* de memória do bloco de memória partilhada, utilizada pela API para implementar a camada de transporte, não se encontra refletido, pelos mesmos motivos supramencionados.

Tabela 5.8: *Overhead* de memória da API de comunicação

	.text	**.data**	**.bss**	**Total**
ISR Handler	4888	4	12	4904
Thread Handler	5060	4	12	5076
No Handler	4744	4	12	4760

5.2 Desempenho

Após a realização das medições dos *footprints* de memória das diferentes versões do *microkernel* e das APIs desenvolvidas, passou-se à realização de testes com o propósito de medir o desempenho destes. A ferramenta utilizada nas medições efetuadas foi a PMU (*Performance Monitoring Unit*). Esta é um componente constituinte do processador ARM Cortex-A9 e será analisado posteriormente nesta secção. Ainda nesta secção serão apresentados os resultados de desempenho obtidos para as versões *singlecore*, SMP, AMP e HMP do *microkernel*, com o intuito de analisar os ganhos a nível do desempenho que as diferentes arquiteturas providenciam. Por fim, as APIs de sincronização e comunicação serão também avaliadas.

5.2.1 ARM *Performance Monitoring Unit*

A PMU pertence à arquitetura da ARM e encontrasse presente no processador ARM Cortex-A9. Esta permite monitorizar vários eventos do processador, sendo assim uma ferramenta bastante útil para reunir estatísticas sobre a operação do mesmo. A PMU presente no Coretex-A9 disponibiliza seis contadores que podem monitorizar qualquer um dos 58 eventos providenciados pelo processador. Estes eventos podem ser desde o número de exceções desencadeadas, número de instruções *load/store* executadas até ao o número de ciclos de relógios decorridos. Nos testes posteriormente apresentados, a PMU é utilizada para contabilizar os ciclos de relógios decorridos na realização destes.

5.2.2 *Microkernel* (*Singlecore* Vs SMP Vs AMP Vs HMP)

Como forma de avaliar o desempenho das versões do *microkernel* desenvolvidas foi realizada uma sequência de testes, com o propósito de avaliar o desempenho destes, para diferentes cargas de trabalho. Nesse sentido, seguidamente é apresentado

os diferentes cenários de testes realizados, seguido da análise e comparação dos resultados obtidos para cada cenário.

Cenários de teste

Os cenários de teste utilizados consistem na execução de um número variável de tarefas, sendo que estas executam todas a mesma sequência de operações, apresentada na listagem 5.1, e foram ajustadas de forma a terem um tempo de execução aproximadamente igual ao período do *tick* do sistema (1ms). Nesse sentido, os testes realizados consistem na avaliação do tempo que cada versão do *microkernel* leva a executar desde 1 a 128 tarefas (incrementos em potências de base 2). Dessa forma é possível avaliar o comportamento, de cada uma das versões, à medida que a carga de trabalho que estas têm de realizar, vai aumentando. Além disso, as versões do *microkernel* testadas foram: (i) *singlcore*; (ii) SMP *quad-core*; (iii) AMP (quad-core); (iv) HMP(3+1) constituído pelo SMP *tri-core* mais *singlecore*; e (v) HMP(2+2) constituído pelo SMP *dual-core* mais AMP *dual-core*.

Contudo, estes cenários de teste não permitem avaliar o impacto que a sincronização e comunicação, entre sistemas operativos, tem no desempenho do sistema. Pois, ao contrário do que acontece na versão SMP, onde estes mecanismos são executados localmente, o mesmo já não é verdade nas versões AMP e HMP, podendo levar a um maior *overhead* na execução destes mecanismos. Por isso, para avaliar os mecanismos posteriormente foram realizados medições de latências para diferentes caso de uso, apresentados nas secções 5.2.3 5.2.4.

```
1  void * thread_test(void *arg)
2  {
3    uint32_t i = 0;
4
5    while(i < 70000){
6      i++;
7    }
8
9    return NULL;
10 }
```

Listagem 5.1: Tarefa utilizada nos testes

1º Teste: 1 Tarefa

O primeiro teste realizado consistiu na execução de apenas uma tarefa. Os resultados obtidos são apresentados na figura 5.1, através de um gráfico de barras, onde cada uma das barras representa o número de ciclos de relógio, que cada versão do *micokernel* necessitou na realização deste teste.

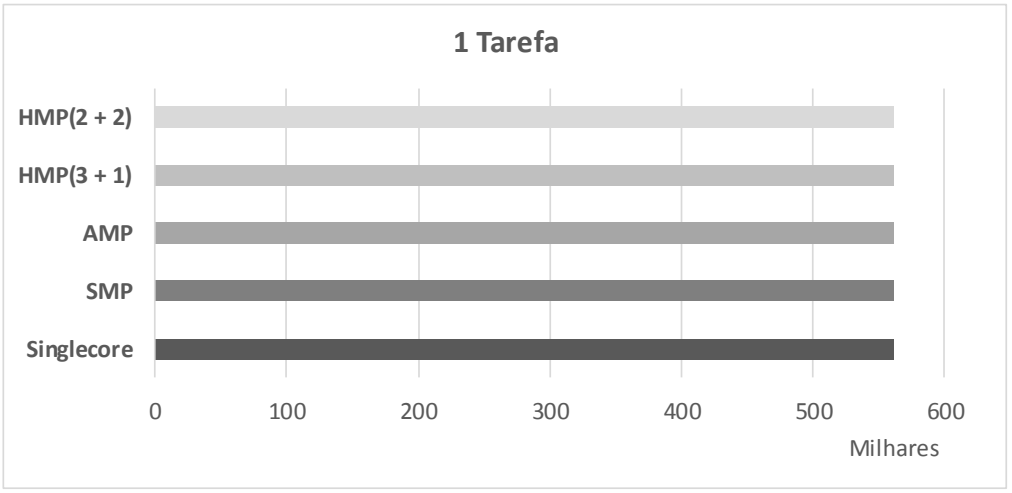

Figura 5.1: Tempos de execução para uma tarefa

Como se pode observar no gráfico apresentado na figura 5.1, os tempos de execução são aproximadamente os mesmos. Pois, como apenas existe uma tarefa para ser executada as versões *multicore* apenas utilizam uma unidade de processamento na execução desta, tal como acontece na versão *singlecore*.

2º Teste: 2 Tarefas

Na figura 5.2 encontrasse apresentado o gráfico referente aos tempos de execução de duas tarefas. Neste teste, já é possível observar diferenças nos tempos de execução. Pois, as versões *multicore* do *microkernel*, pelo facto de utilizarem mais unidades de processamento do que a versão *singlecore*, conseguem executar as duas tarefas em simultâneo.

Contudo, ao contrário do que acontece nas versões AMP e nas duas variantes do HMP, onde o tempo de execução manteve-se igual ao do teste anterior, a versão SMP apresentou um aumento no tempo de execução. Este aumento deve-se ao funcionamento do escalonador da versão SMP pois, este apenas seleciona uma

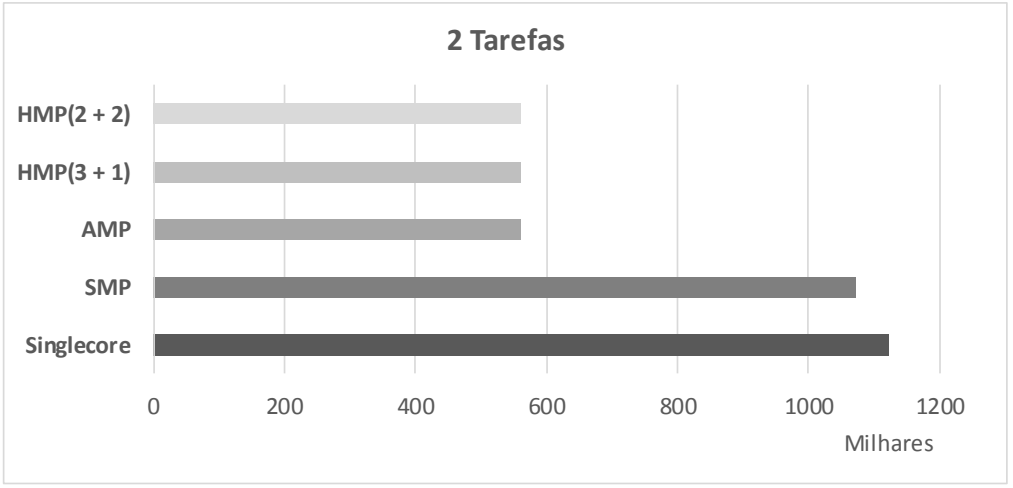

Figura 5.2: Tempos de execução para duas tarefas

tarefa para executar a cada *tick* do sistema, levando a que a versão SMP só comece a executar tarefas concorrentemente a partir do primeiro *tick* do sistema.

3º Teste: 4 Tarefas

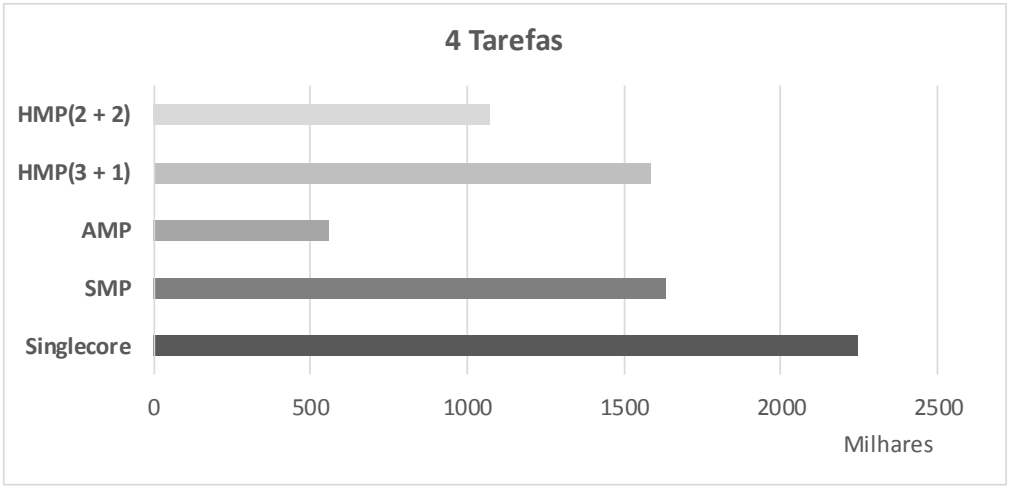

Figura 5.3: Tempos de execução para quatro tarefas

Como se pode observar na figura 5.3, à medida que o número de tarefas aumenta também a diferença entre as versões *multicore* e a *singlecore* aumenta. Além disso, a versão AMP ainda não sofreu nenhum aumento no tempo de execução pois, esta é capaz de lançar quatro tarefas simultaneamente, uma por cada cópia do *micro-*

kernel. No caso do HMP(3+1), como este só possui duas cópias do *microkernel* (SMP *tri-core* mais *singlecore*) este não é capaz de lançar, na sua fase inicial, quatro tarefas simultaneamente, devido ao funcionamento do escalonador da cópia SMP, explicado anteriormente. Isto aproxima o seu tempo de execução ao da versão SMP. No caso do HMP(2+2), como este consegue lançar três tarefas em simultâneo, para este teste este é menos penalizado pelo funcionamento do escalonador do *microkernel* SMP, apresentando assim um melhor desempenho que a variante HMP(3+1).

4º Teste: 8 Tarefas

Na figura 5.4 estão expostos os resultados obtidos na execução de oito tarefas. Neste teste já se verifica um aumento do tempo de execução do AMP, pois cada cópia do *microkernel* constituinte deste já necessita de executar duas tarefas.

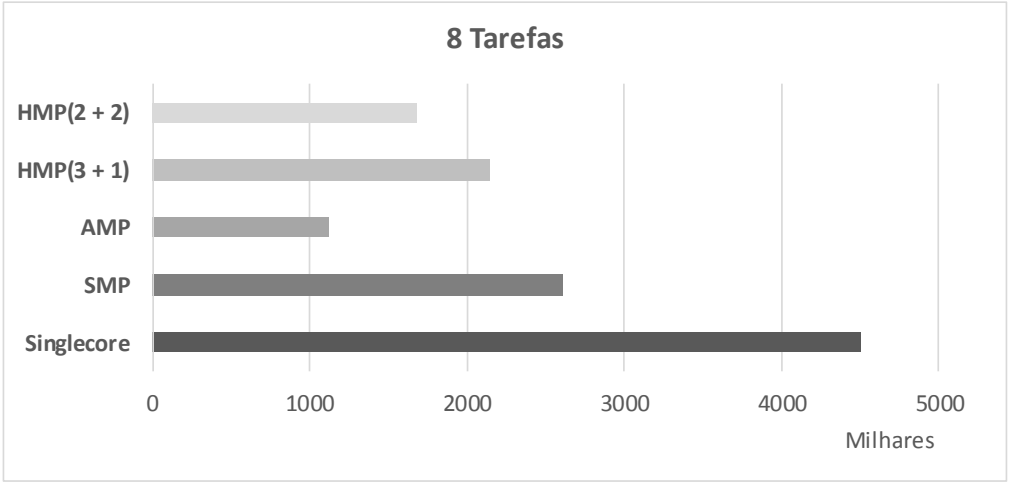

Figura 5.4: Tempos de execução para oito tarefas

5º Teste: 16 Tarefas

Na figura 5.5 estão apresentados os resultados dos tempos de execução, referentes à execução de dezasseis tarefas.

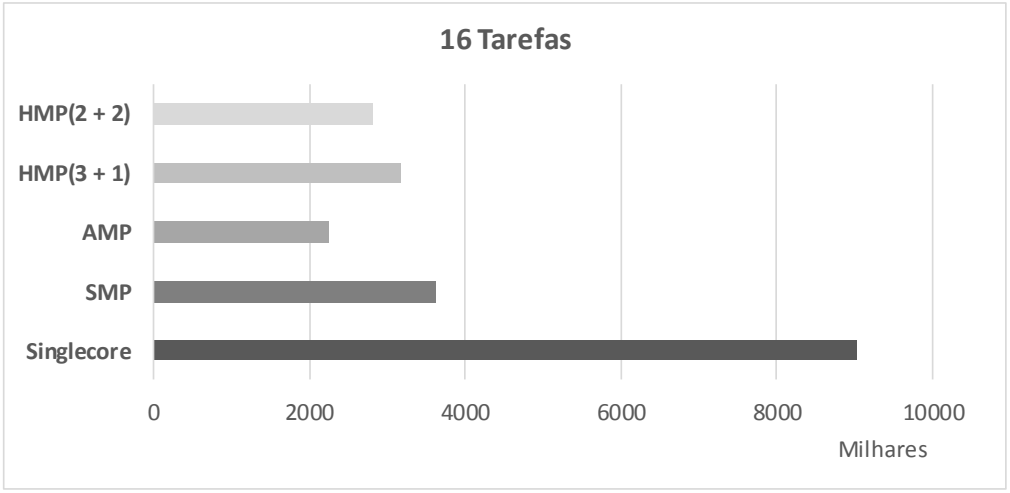

Figura 5.5: Tempos de execução para dezasseis tarefas

6º Teste: 32 Tarefas

Na figura 5.6 estão apresentados os resultados dos tempos de execução, referentes à execução de trinta e duas tarefas.

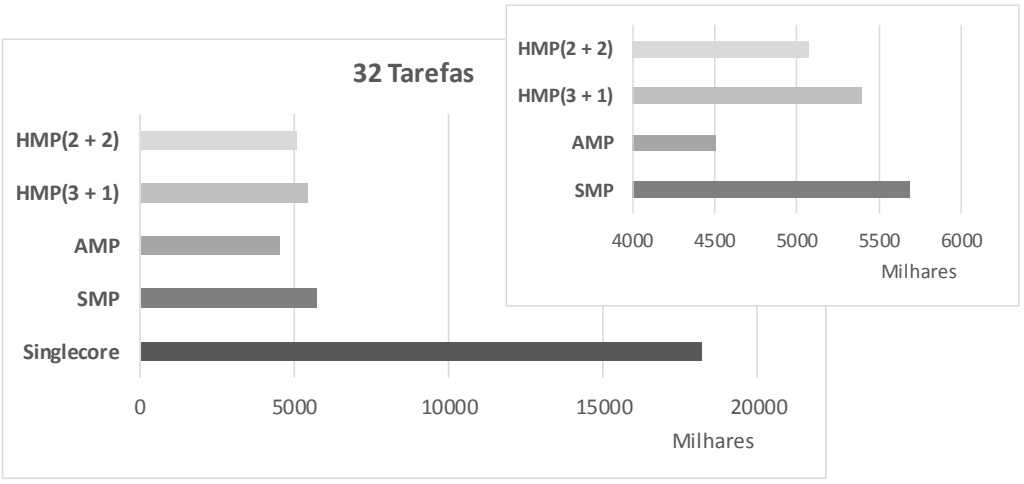

Figura 5.6: Tempos de execução para trinta e duas tarefas

Tendo em consideração os testes apresentados anteriormente e o presente teste, é possível constatar que a versão AMP apesar de apenas no 4º teste ter apresentado um aumento no tempo de execução, este agora tem apresentado um aumento no tempo de execução proporcional ao aumento do número de tarefas. Sendo este um aumento superior ao verificado pelas versões SMP e HMP. Além

disso, o HMP(2+2) demonstra um melhor desempenho comparativamente com o HMP(3+1), devido à menor sobrecarga de cada imagem do *microkernel*.

7º Teste: 64 Tarefas

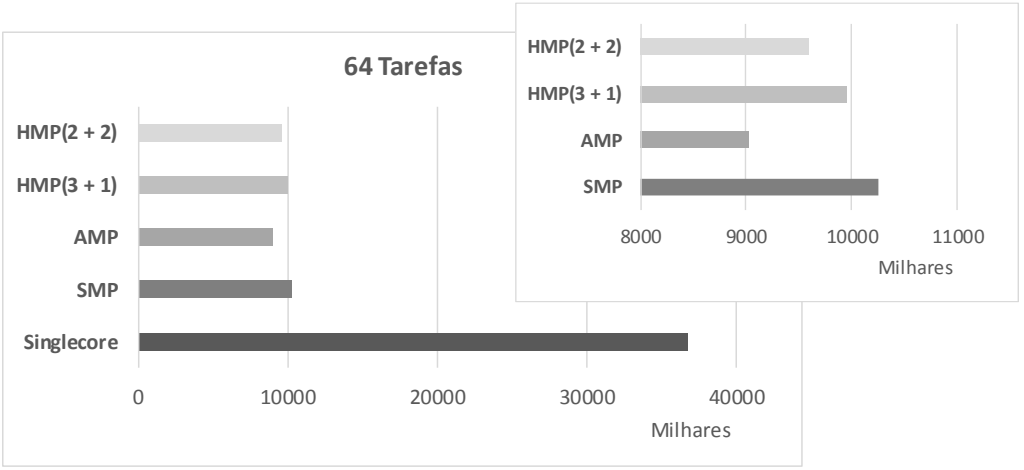

Figura 5.7: Tempos de execução para sessenta e quatro tarefas

Na figura 5.7 estão apresentados os resultados dos tempos de execução, referentes à execução de sessenta e quatro tarefas.

8º Teste: 128 Tarefas

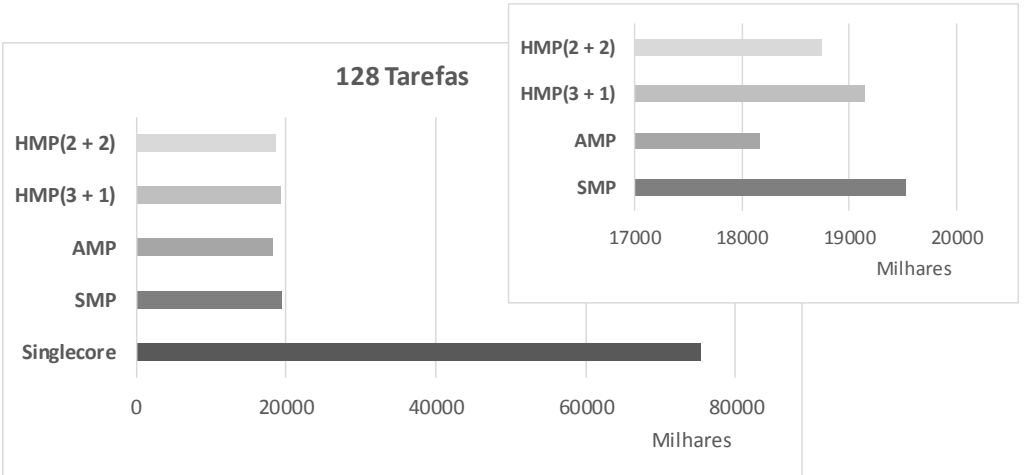

Figura 5.8: Tempos de execução para cento e vinte e oito tarefas

No gráfico exposto na figura 5.8 estão apresentados os valores dos tempos de execução para 128 tarefas. Neste é possível constatar que as versões *multicore* apresentam um desempenho muito mais elevado quando comparadas com a versão *singlecore*, para cargas de trabalho superiores. Comparando as versões *multicore* entre si, pode-se concluir que as versões que utilizam um maior número de imagens do *microkernel* também apresentam melhor desempenho. Contudo, mais uma vez é importante referir, que os testes realizados não têm em consideração o *overhead* introduzido pela sincronização e comunicação entre sistemas operativos, que as versões AMP e HMP necessitam.

Comparação dos resultados obtidos

Na figura 5.9, são apresentados todos os valores de desempenho recolhidos para as versões *multicore* do *microkernel* num gráfico de escala logarítmica de base 2, de forma a possibilitar uma melhor comparação entre o desempenho das diferentes arquiteturas utilizadas.

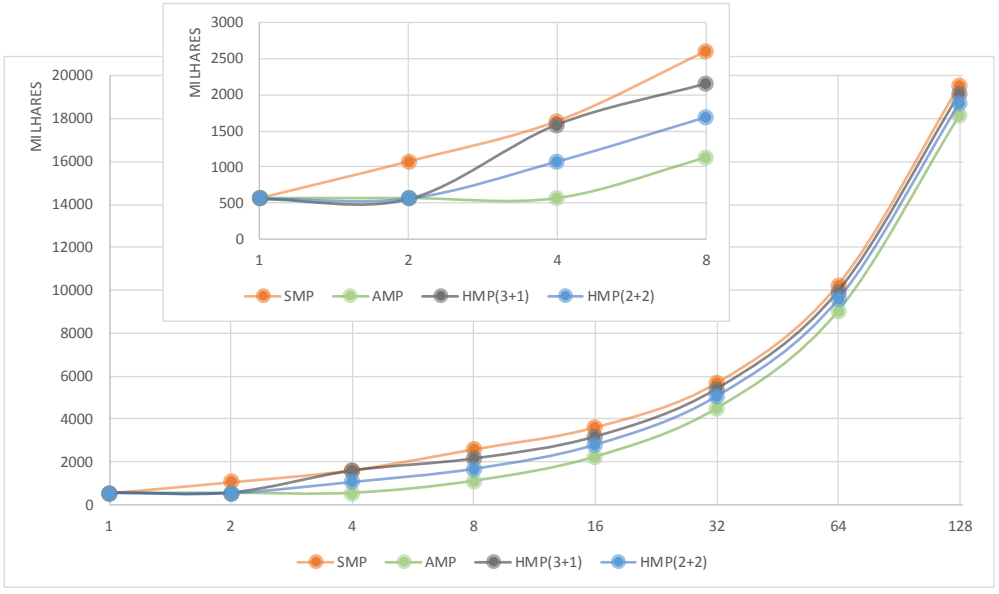

Figura 5.9: Comparação dos tempos de execução das versões *multicore*

5.2.3 API de sincronização

A medição das latências das funções disponibilizadas pela API de sincronização foram condicionadas pelo facto de estas necessitarem de funcionalidades disponibilizadas pelos sistemas operativos, principalmente no que diz respeito à suspensão e resumo de tarefas. Nesse sentido, na tabela 5.9 são apresentados os valores de latência obtidos em ciclos de relógio, para as diferentes funções disponibilizadas pela API, juntamente com a descrição do cenário para o qual foi realizada a medição.

Tabela 5.9: Valores de latência da API de sincronização

Função	**Latência**	**Cenário**
`global_mutex_lock`	63	Aquisição do *mutex* global quando este se encontra livre.
`global_mutex_unlock`	105	Libertação do *mutex* global e não existe nenhuma tarefa bloqueada neste.
`global_mutex_trylock`	41	Função independente do sistema operativo. Tenta adquirir o *mutex* global.
`global_sem_post`	113	Incrementação do *semaphore* global quando não existe tarefas em espera neste.
`global_sem_wait`	66	Decremento do *semaphore* global quando o valor deste é superior a 0.
`global_get_value`	39	Leitura do valor atual do contador do *semaphore* global.

5.2.4 API de comunicação

Como forma de avaliar o desempenho, da API de comunicação entre sistemas operativos, foram realizados testes de forma a avaliar a latência no envio de mensagens, com diferentes tamanhos, e posteriormente na receção destas. A comunicação nos testes efetuados foi configurada para operar no modo *no handler* pois, este não necessita de suporte do sistema operativo garantindo assim, que a latência da comunicação não se encontra condicionada pelo sistema operativo.

Tabela 5.10: Tempos de envio e receção de mensagens

Função	**1 byte**	**4 bytes**	**16 bytes**	**64 bytes**	**256 bytes**
`msg_send`	300	300	354	558	1374
`msg_recv`	198	280	480	1467	5040

Na tabela 5.10 são apresentados os valores, em ciclos de relógio, da latência da comunicação, no envio e receção de mensagens de diferentes dimensões. Os valores medidos permitem concluir, que o processo de envio de mensagens é, em média, mais rápido do que o processo de receção. Isto deve-se ao facto de os *buffers* utilizados para o envio de mensagens possuírem dimensões fixas, neste caso 256 *bytes*, possibilitando que a mensagem seja copiada, do espaço do utilizador para os *buffers* de envio, em blocos de 4 *bytes*, o que explica também o facto de o envio de 1 *byte* demorar o mesmo tempo que o envio de 4 *bytes*. No entanto, este mecanismo já não pode ser aplicado no processo *inverso* pois, caso as dimensões do *buffer*, disponibilizado pelo utilizador, não for múltiplo de 4 existe o risco de corromper o espaço de memória do utilizador.

Capítulo 6

Conclusões

Neste último capítulo da dissertação, são apresentadas as conclusões retiradas do trabalho realizado. Além disso, são apresentadas propostas que visam a expansão e melhoramento do trabalho realizado.

6.1 Conclusão

A presente dissertação apresenta o *porting* de um *microkernel* SMP para duas plataformas de desenvolvimento baseadas na arquitetura ARM Cortex-A9 MPcore, e posterior *refactoring* deste para as arquiteturas de multi-processamento AMP e HMP. Além disso, foram também apresentadas as implementações das APIs de suporte às novas versões do *microkernel*. Por fim, foi ainda explorada a expansão do HMP para uma versão heterogénea denominada de H^2MP.

Este projeto foi sem dúvida desafiante pela variedade de conhecimentos envolvidos na implementação deste. Desde a compreensão da arquitetura de processadores ARMv7-A, passando pelos sistemas operativos (essencialmente sistemas operativos baseados em *microkernels* com suporte ao multi-processamento), diferentes arquiteturas de multi-processamento, APIs de comunicação entre *cores* (*inter-core communication*), linguagem *assembly* e a exploração das ferramentas de desenvolvimento (Xilinx ISE Design Suite e ARM Fast Models).

Relativamente aos objetivos estipulados, estes foram efetivamente cumpridos. Na fase inicial do trabalho foi empregue um esforço considerável na análise e compreensão do funcionamento e implementação do ARM *microkernel* SMP. Contudo,

esta análise morosa e minuciosa permitiu a aquisição de conhecimento indispensáveis para as fazes posteriores do trabalho, visto que estas requereram alterações de mecanismos cruciais do *microkernel*. Após a conclusão desta análise inicial, foi realizado o *porting* do *microkernel* para as plataformas Zynq-7000 e VE. Depois, foi realizado o *refactoring* do *microkernel* para a versão AMP, passando inicialmente pelo *refactoring* do *microkernel* para a versão *singlecore*, necessária para a construção do sistema AMP. O objetivo seguinte, e também o mais desafiante, consistiu na implementação do sistema HMP, conjugando as versões SMP e AMP, e expansão deste para um modelo heterogéneo, combinando no mesmo sistema o *microkernel* HMP desenvolvido juntamente com o sistema operativo de tempo real FreeRTOS. Além disso, para a realização deste objetivo foram também implementadas duas APIs de suporte: uma para possibilitar a sincronização entre os sistemas operativos; e outra para permitir o envio de mensagens entre os mesmos. Relativamente aos resultados experimentais obtidos, estes permitem concluir que a nova abordagem de multi-processamento denominada de HMP, desenvolvida na presente dissertação, apresenta um bom compromisso entre o *overhead* de memória e o desempenho, em comparação às versões SMP e AMP. Este é capaz de providenciar um ambiente de execução *multicore*, semelhante ao SMP, em paralelo com um ambiente de execução *singlecore*, semelhante ao providenciado no AMP.

6.2 Trabalho Futuro

Apesar dos objetivos propostos na presente dissertação terem sido cumpridos, existem melhorias que podem ser concretizadas de forma a expandir o trabalho realizado.

Assim, a primeira sugestão recai sobre a realização dos teste realizados no capítulo 5 na plataforma de desenvolvimento Zynq-7000. No entanto, como esta apenas possui duas unidades de processamento é também sugerido a realização do *porting* para uma plataforma iMX6, que pelo facto de esta já possuir um processador *quad-core*, possibilita a realização de testes às arquiteturas HMP e H^2MP, que não são passiveis de serem realizados na Zynq-7000. Além disso, é também sugerido a realização de mais testes, se possível com recurso a *benchmarks* (embora não seja fácil dada a escassez/inexistência de *benchmarks* para *multicore*).

A segunda sugestão consiste na expansão da API de comunicação entre sistemas operativos, adicionando suporte para diferentes tipos de comunicação, para além

do envio de mensagens. Como exemplo, podem ser implementados *Packet Channels*, *Scalar Channels* e um mecanismo de envio de mensagens por referência (*Zero Copy*).

A terceira sugestão relaciona-se com a gestão da configurabilidade do *microkernel*. Neste momento, a configuração do *microkernel* é feita com recurso à compilação condicional, sendo assim necessário a configuração de um conjunto de *macros*, de acordo com a configuração do *microkernel* pretendida. Nesse sentido, é proposto a utilização do GME (*Generic Modeling Environment*) para implementar interface gráfica, capaz de gerir toda a configurabilidade do sistema, desde a arquitetura do *microkernel* até à escolha dos sistemas operativos integrantes do sistema H^2MP.

A quarta e última sugestão propõe a integração de um sistema operativo de propósito geral como o Linux no sistema H^2MP, utilizando as APIs de sincronização e comunicação na integração deste. Dada a necessidade dos sistemas embebidos atuas terem um sistema GUI de monitorização ou de interação com o utilizador, conjuntamente com serviços de tempo-real.

Bibliografia

[1] M. Graphics, "Nucleus RTOS." [Online]. Available: http://www.mentor.com/embedded-software/nucleus/

[2] Q. S. Systems, "Operating systems, development tools, and professional services for connected embedded systems." [Online]. Available: http://www.qnx.com/

[3] Xilinx, "Zynq-7000 All Programmable SoC Technical Reference Manual," p. 1862, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[4] ——, "Zynq-7000 All Programmable SoC Overview," pp. 1–21, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[5] A. Limited, "Fixed Virtual Platforms," pp. 1–49, 2014.

[6] D. Andrews, I. Bate, T. Nolte, C. M. Otero-Perez, and S. M. Petters, "Impact of Embedded Systems on RTOS Use and Design," *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 13–20, 2005.

[7] B. Moyer, *Real World Multicore Embedded Systems*. Elsevier Inc., 2013.

[8] J. Mistry, M. Naylor, and J. Woodcock, "Adapting FreeRTOS for Multicore: an Experience Report," University of York, Tech. Rep., 2010.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. John Wiley & Sons Inc., 2009.

[10] A. Limited, *ARM Cortex -A Series*. ARM Limited, 2013.

[11] Q. S. Systems, "Running AMP , SMP or BMP Mode for Multicore Embedded Systems QNX Software Systems," *Beyond Bits*, pp. 92–97, 2012.

[12] G. Heiser, "The role of virtualization in embedded systems," *Proceedings of the 1st workshop on Isolation and integration in embedded systems IIES 08*, pp. 11–16, 2008.

[13] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide For Engineers And Programmers*, 2005.

[14] B. Roch, "Monolithic kernel vs . Microkernel."

[15] A. N. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Elsevier Inc., 2004.

[16] C. Wang, B. Yao, Y. Yang, and Z. Zhu, "A Survey of Embedded Operating System," 2001.

[17] R. Oshana and M. Kraeling, *Software Engineering for Embedded Systems*. Elsevier Inc., 2013.

[18] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

[19] P. Parkinson, "Safety, security and multicore," *Advances in Systems Safety - Proceedings of the 19th Safety-Critical Systems Symposium, SSS 2011*, pp. 215–232, 2011.

[20] J. Mistry, "FreeRTOS and Multicore," Ph.D. dissertation, University of York, 2011.

[21] S. Nagarajan and N. Vulpe, "Processor Affinity or Bound Multiprocessing ?" 2009.

[22] M. Embedded, "Nucleus RTOS," p. 5, 2012.

[23] L. Qnx and J. Lennox, "QNX vs. L4," NICTA, Tech. Rep., 2008.

[24] M. P. Division, "ARM MPCore Boot and Synchronization Example Code," pp. 1–33, 2009.

[25] J. Langbridge, *Professional Embedded ARM Development*. John Wiley & Sons, Inc., 2013.

[26] A. Limited, *ARM Generic Interrupt Controller Architecture Specification*. ARM Limited, 2008. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0048a/IHI0048A_gic_architecture_spec_v1_0.pdf

[27] ——, *ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.